Universidad Nacional Mayor de San Marcos

Universidad del Perú. Decana de América

Facultad de Ingeniería de Sistemas e Informática

Escuela Profesional de Ingeniería de Software

Implementación de nuevos algoritmos de las listas estáticas basado en las propiedades de los algoritmos genéticos, que demuestre que las listas estáticas son más eficientes en tiempo de gestión de datos que las listas dinámicas

TESIS

Para optar el Título Profesional de Ingeniero de Software

AUTOR

Hernán Elías CHIRA HUAMÁN

ASESOR

María Elena RUIZ RIVERA

Lima, Perú

2019

Referencia bibliográfica

Chira, H. (2019). *Implementación de nuevos algoritmos de las listas estáticas basado en las propiedades de los algoritmos genéticos, que demuestre que las listas estáticas son más eficientes en tiempo de gestión de datos que las listas dinámicas*. [Tesis de pregrado, Universidad Nacional Mayor de San Marcos, Facultad de Ingeniería de Sistemas e Informática, Escuela Profesional de Ingeniería de Software]. Repositorio institucional Cybertesis UNMSM.

#182

UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
Universidad del Perú. DECANA DE AMÉRICA
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
ESCUELA PROFESIONAL DE INGENIERÍA DE SOFTWARE

Acta de Sustentación de Tesis

Siendo las 11:00 del día 25 de enero del año 2019, se reunieron los docentes designados como miembros de Jurado de la Tesis, presidido por el Lic. John Ledgard Trujillo Trejo, Lic. Pablo Jesús Romero Naupari (Miembro), y la Lic. María Elena Ruiz Rivera (Miembro Asesor) para la sustentación de la Tesis intitulada: "IMPLEMENTACIÓN DE NUEVOS ALGORITMOS DE LAS LISTAS ESTÁTICAS BASADO EN LAS PROPIEDADES DE LOS ALGORITMOS GENÉTICOS, QUE DEMUESTRE QUE LAS LISTAS ESTÁTICAS SON MÁS EFICIENTES EN TIEMPO DE GESTIÓN DE DATOS QUE LAS LISTAS DINÁMICAS"; por el bachiller **Hernán Elías Chira Huamán**, para optar el Título Profesional de Ingeniero de Software.

Acto seguido de la exposición de la Tesis, el Presidente invitó al bachiller a dar respuesta a las preguntas establecidas por los Miembros de Jurado.

El bachiller en el curso de sus intervenciones demostró pleno dominio del tema, al responder con acierto y fluidez a las observaciones y preguntas formuladas por los señores miembros del Jurado.

Finalmente habiéndose efectuado la calificación correspondiente por los miembros de Jurado, la bachiller obtuvo la nota de 17 (En letras) Diez y siete.

A continuación el Presidente del Jurado, John Ledgard Trujillo Trejo declara al Bachiller **Ingeniero de Software**.

Siendo las 17:50 horas, se levantó la sesión.

.....
Lic. John Ledgard Trujillo Trejo
Presidente

.....
Lic. Pablo Jesús Romero Naupari
Miembro

.....
Lic. María Elena Ruiz Rivera
Miembro Asesor

FICHA CATALOGRÁFICA

CHIRA HUAMÁN, Hernán Elías

IMPLEMENTACIÓN DE NUEVOS ALGORITMOS DE LAS LISTAS ESTÁTICAS BASADO EN LAS PROPIEDADES DE LOS ALGORITMOS GENÉTICOS

Implementación de algoritmos
(Lima, Perú 2019)

Tesis, Facultad de Ingeniería de Sistemas, Universidad Nacional Mayor De San Marcos

DEDICATORIA:

Este trabajo está dedicado a mi madre que siempre ha estado apoyándome durante toda mi vida.

AGRADECIMIENTOS

A la profesora María Elena Ruiz Rivera, por su orientación para que este trabajo cumpla con los objetivos trazados.

A la UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS por darme la oportunidad de estudiar y ser un profesional.

Agradecerle a Dios por bendecirme para llegar hasta donde he llegado.

RESUMEN

El presente trabajo de investigación tiene por finalidad aplicar la implementación de las operaciones de listas dinámicas en base a registros denominados listas estáticas.

A su vez, demostrar que las operaciones en listas estáticas pueden ser semejantes o más eficientes en las inserciones o eliminaciones en tiempo que las listas dinámicas. Las listas dinámicas podrán ser más eficientes que los registros en la utilización y ahorro de espacio de memoria, pero aun así demanda más tiempo en almacenar los datos uno por uno al crear nuevos nodos, e inclusive puede haber problemas de interrupción del compilador en la gestión de datos. Las listas estáticas permiten una mayor eficiencia en gestión de menores cantidades de datos en comparación de las listas dinámicas.

Finalmente, proponiendo la creación de algoritmos en el uso de inserción, eliminación, búsqueda, guardado y recuperado de datos de listas estáticas, se debe expandir su respectivo uso lógico y expandir sus características de implementación lógica de las demás estructuras de datos dinámicos, permitiendo múltiples aplicaciones en el rendimiento de control de tiempo y estabilidad del espacio de memoria en la gestión de datos.

Todo esto constituye el aporte que deseo brindar al desarrollo de la programación y aplicaciones múltiples de la estructura de datos.

Palabras claves: listas dinámicas, listas estáticas, inserción de listas, eliminación de listas, búsqueda de listas, registro, nodo, tiempo de listas, datos en listas, lógica de nodo, listas y genética.

ABSTRACT

This research work aims to apply the implementation of the operations of dynamic lists based on records called static lists.

In turn, show that static lists operations may be similar or more efficient insertions or deletions in time dynamic lists. Dynamic lists may be more efficient than the records in the use and saving memory space, but still takes longer to store the data one by one to create new nodes, and even there may be problems interruption compiler management of data. Static lists allows more efficient management of smaller amounts of data than dynamic lists.

Finally, proposing the creation of algorithms using insert, delete, search, saved and retrieved data from static lists, it should expand their respective logical use and expand its features logical implementation of other dynamic data structures, allowing multiple applications in performance time control and the memory space stability in data management.

All this is the contribution I wish to give to the development of multiple applications programming and data structure.

Keywords: dynamic lists, static lists, list insertion, list deletion, list search, record, node, list time, list data, node logic, lists and genetics.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

En las primeras apariciones de las computadoras digitales, la programación de lenguajes máquina accedía a una tabla específica de datos usando los vectores y matrices, además de la aplicación de cálculos matemáticos en ellos. Los lenguajes ensambladores no podían emular las características de una determinada cantidad de almacenamiento de datos. Los lenguajes de alto nivel que apoyaron la aplicación de vectores fueron Fortran, Cobol y Algol 60. Seguidamente, C y C++ actualizaron y mejoraron la aplicación de vectores a través de la definición de su tipo de datos.

Los registros poseen múltiples campos. Cada uno de estos campos tiene un específico nombre y tipo de dato. A comparación de los vectores que solo poseen un único nombre y único tipo de dato. Los registros ayudan a la correcta, ordenada y eficiente gestión pequeña o masiva de datos. COBOL fue uno de los primeros lenguajes que apoyó la aplicación de registros, seguido de Pascal. Luego aparecieron los lenguajes de C, Ada, Modula y Java, en la cual mejoraron la aplicación de registros asemejándose más al lenguaje humano.

La aplicación de las listas enlazadas se convirtió como la primera estructura de datos en el lenguaje de procesamiento de información (IPL), para el desarrollo de programas inteligentes. Posteriormente, estas listas fueron usadas en el lenguaje de programación LISP como una de sus estructuras de datos principales en sus aplicaciones de código fuente.

El beneficio de construir listas enlazadas, permite que los elementos de la lista pueden ser fácilmente insertado o eliminado sin desperdiciar memoria, porque no existe almacenamiento contiguo y no se desperdicia memoria como en los vectores.

Al contrastar las listas enlazadas con los vectores podemos observar que a diferencia de los vectores, las listas enlazadas utilizan más operaciones aritméticas en los punteros. Los punteros son también llamados enlaces y conectores porque poseen un puente de comunicación lineal entre un determinado conjunto de datos. Dichos punteros desperdician más tiempo al tratar de conectar los enlaces de los nodos de la lista dinámica.

CAPÍTULO I. PLANTEAMIENTO METODOLÓGICO

1.1 Antecedentes del problema

Normalmente las aplicaciones de listas dinámicas han sido muy usadas de distintas formas y lógicas en las distintas operaciones por diferentes autores en diferentes lenguajes como insertar antes o después de una lista de datos, eliminar antes o después de una lista de datos, buscar algún dato, etc; en base al insertar inicio o final de la lista de datos, y eliminar inicio o final de la lista de datos. Pero hasta la fecha no hubo una aplicación que permita la implementación de estas características de las listas dinámicas en base a estructuras estáticas y determinar si es una buena opción en la gestión de una cierta cantidad de datos en un tiempo menor o igual que al de las listas dinámicas.

La importancia de la implementación de las operaciones en listas estáticas sirve como una aplicación de mejora en tiempo o semejante al control de las listas dinámicas, en las gestión de miles de datos, uniendo las características lógicas de implementación de listas dinámicas y en base al uso de registros en la simulación de los nodos dinámicos.

1.2 Definición del problema

La problemática de la gestión de datos en la estructura de datos no solo se puede organizar con la estructura de datos dinámicas por ser más eficiente en el control de espacio de memoria.

La estructura de datos dinámicos también tienen problemas al crear nuevo espacio para un registro de datos y demanda más tiempo en la inserción, eliminación, búsqueda de grandes cantidades de datos; inclusive hay problema de corte de compilador por el uso alterado de la memoria. Un ejemplo es el caso de trabajar con las operaciones de inserción y eliminación de árboles, no soporta grandes cantidades de datos en el compilador de C en el uso de memoria estándar.

Las listas estáticas sirven como referencia de apoyo en la gestión de datos con la implementación de las características lógicas de las listas dinámicas, y ayudan en la gestión de datos eficiente en un tiempo similar o menor a la gestión de datos de manera dinámica.

Adicionalmente, se debe proporcionar la aplicación y características de las diferentes y múltiples estructuras dinámicas tales como pilas, colas, árboles a la base de las estructuras estáticas. Con ello se puede controlar sus ventajas o desventajas en cada comparación de éstas estructuras.

1.3 Objetivos

1.3.1 Objetivo principal

- Implementar algoritmos para listas estáticas o listas con registro en base a las operaciones lógicas de las listas dinámicas.

1.3.2 Objetivos secundarios

- Demostrar que el tiempo de inserción, eliminación, búsqueda, guardado de datos es semejante o menor en tiempo de gestión de datos en las listas estáticas que en las listas dinámicas.

- Implementar un algoritmo propio de listas dinámicas que tenga un menú de inserción, eliminación, búsqueda, guardado en la gestión de datos.

- Realizar el algoritmo de listas estáticas en base a la lógica del funcionamiento y comportamiento del nodo en la lista dinámica.

- Proponer el uso de pilas, colas en las listas estáticas y dinámicas conjuntamente con sus aplicaciones: simples, dobles, circulares y doblemente circulares.

- Desarrollar una integración entre las listas estáticas y la gestión de fases para los algoritmos genéticos.

1.4 Justificación

Implementar las distintas operaciones de listas estáticas en la codificación en base a registros es un buen sustituto de las operaciones de las listas dinámicas al contar con miles de datos y no solo tener una prioridad de millones de datos en un solo registro universal de gestión de datos. No solamente las listas dinámicas tienen un eficiente control de miles de datos en la gestión de datos, también las listas estáticas pueden ser de igual o mejor tiempo y es más estable en el control predeterminado de espacio.

Implementar listas estáticas es una buena y alternativa versión de implementar listas dinámicas ya que sirve como una introducción para el entendimiento de las listas dinámicas, las operaciones de listas estáticas gira en torno de la creación del nodo para un mayor entendimiento de las listas dinámicas.

Los beneficios de la investigación es tratar de analizar diferentes implementaciones de las listas dinámicas de diversos autores con sus pros y contras, luego adaptar al propio algoritmo de listas dinámicas a que contenga todas las operaciones posibles de gestión de los datos en un solo menú. Normalmente la mayoría de los autores explican a detalle las funcionalidades y características de su algoritmo, pero por separado como la inserción y eliminación entre el conjunto de datos, y no lo implementan en un menú universal. Ciertos autores si usan un menú universal para la gestión de datos, pero no explican a detalle el procedimiento de cada paso de sus funciones de gestión de datos.

El propósito final de investigación en base a esta problemática es proponer un menú universal de la gestión de datos tanto para listas dinámicas como listas estáticas, y explicar detalladamente cómo realizar los procesos lógicos en cada función y operación de la gestión de datos en base a mis reglas generales.

1.5 Propuesta metodológica

La solución comprende lo siguiente: creación de algoritmos en el uso de inserción, eliminación, búsqueda, editado, guardado y recuperado de datos de listas estáticas, con sus respectivas características de implementación lógica, permitiendo múltiples aplicaciones en el rendimiento de control de tiempo y estabilidad del espacio de memoria en la gestión de datos.

Adicionalmente, hacer comparaciones y análisis en las operaciones de listas estáticas con listas dinámicas en función tiempo con una determinada cantidad de datos y/o registros.

1.6 Organización de la tesina

La presentación del resto de la tesina está organizada de la siguiente forma:

En el Capítulo II, se describirá brevemente los conceptos básicos de estructura de datos, las funciones principales de listas enlazadas tales como el recorrido de la lista, la inserción de un dato específico en un lugar específico de la lista enlazada, la eliminación de un dato específico en un lugar específico de la lista enlazada, para un mejor entendimiento de la presente tesis.

En el Capítulo III se abordará el estado de arte, con el análisis de métodos y técnicas de algoritmos de las listas enlazadas para el análisis, especificación, críticas, observaciones y mejoramiento respectivo del propio algoritmo a implementar.

En el Capítulo IV, se describirá el desarrollo de la solución, presentando las funciones implementadas que serán utilizadas en el desarrollo y funcionamiento del sistema del menú de gestión de datos para resolver el problema descrito. También se presentarán las pruebas necesarias en distintas iteraciones sea la cantidad de datos y/o registros de datos para el control de tiempo en la comparación de las listas estáticas y listas dinámicas. Finalmente, se hará una aplicación entre una integración del gestionamiento de datos de las fases de los algoritmos genéticos usando las operaciones de las listas estáticas.

En el Capítulo V, se presentarán las conclusiones de este trabajo y recomendaciones.

Finalmente, presentaré las referencias bibliográficas que nos han servido para la realización de la presente tesis.

CAPÍTULO II. MARCO TEÓRICO

2.1 Conceptos Básicos de Estructura de Datos

2.1.1 Vector

Es un tipo de datos estructurado compuesto de un número determinado de elementos, de tamaño fijo y elementos homogéneos (del mismo tipo). La característica de tamaño fijo se refiere a que el tamaño del vector debe ser conocido en tiempo de compilación. La representación gráfica del vector se muestra de la siguiente manera:

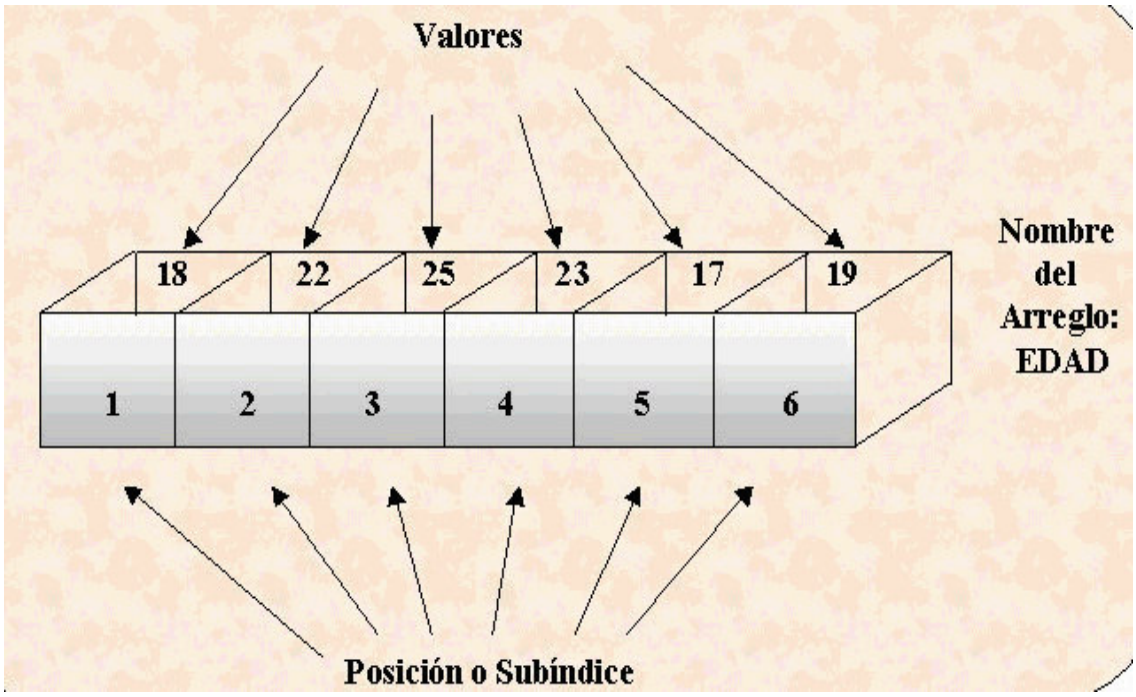

Figura 2.1: Vector de las edades de distintas personas. [1]

2.1.2 Registro

Son estructuras de datos cuyos elementos pueden ser de diferentes tipos. En los registros se puede almacenar una variedad de información sobre una persona (nombre, estado civil, edad, fecha de nacimiento, etc.). La representación gráfica del registro se muestra de la siguiente manera:

Figura 2.2: Registro de matrícula de curso del alumno. [2]

2.1.3 Puntero

Un puntero es una variable que contiene la dirección de memoria de un dato o de otra variable que contiene al dato en un arreglo.

Los punteros permiten simular el paso por referencia, crear y manipular estructuras dinámicas de datos, tales como listas enlazadas, pilas, colas y árboles. La representación gráfica del puntero se muestra de la siguiente manera:

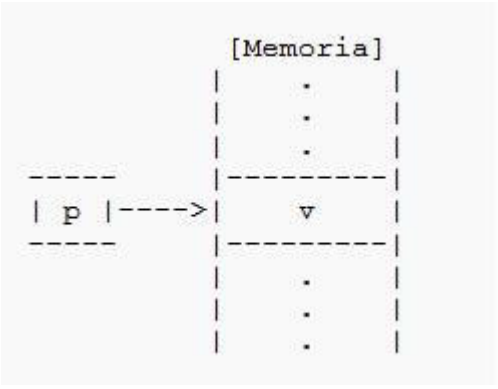

Figura 2.3: Representación gráfica de un puntero "p" y su dirección de memoria "v". [3]

2.1.4 Nodo

En estructuras de datos, un nodo es uno de los elementos de una lista enlazada, de un árbol o de un grafo.

Cada nodo será una estructura o registro que dispondrá de varios campos, y al menos uno de esos campos será un puntero o referencia a otro nodo, de forma que, conocido

un nodo, a partir de esa referencia, será posible en teoría tener acceso a otros nodos de la estructura.

Los nodos son herramientas esenciales para uno de los procesadores que lo componen. La representación gráfica del nodo simple y nodo doble se muestra de la siguiente manera:

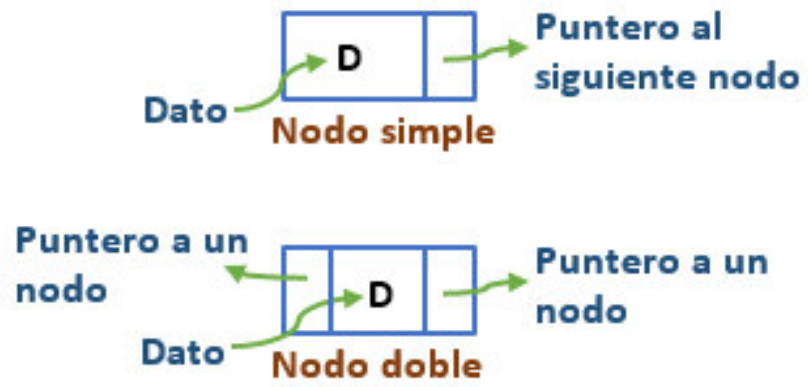

Figura 2.4: Representación gráfica de un nodo simple y doble. [4]

2.1.5 Campo

En términos de aplicaciones de programación, es el espacio reservado para introducir determinados datos asociados a una categoría de clasificación. El contenido de cada fila por campo se muestra de la siguiente manera:

Field Name	Data Type
id_usuario	int
id_perfil	int
nombre	varchar(100)
contrasenia	varbinary(8000)
id_personal	bigint
foto	image
activo	bit

Figura 2.5: Atributos con nombre de campo y tipo de dato declarado usando un respectivo espacio reservado. [5]

2.1.6 Lista enlazada o lista dinámica

Colección de Nodos lógicamente uno detrás de otro. Cada nodo se compone al menos de 2 campos: La información propiamente dicha y un puntero al próximo elemento de la lista. La representación gráfica de la lista enlazada se muestra de la siguiente manera:

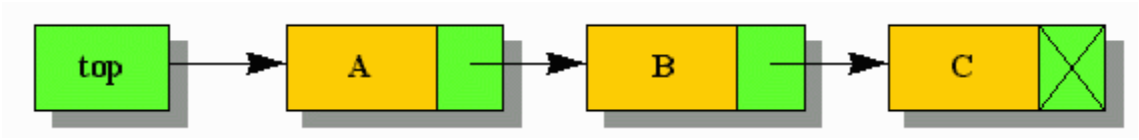

Figura 2.6: Representación gráfica de un conjunto de elementos(A, B, C), sus respectivos punteros (flechas y marca X) y la cabeza de inicio "top" de una lista enlazada. [6]

Se tiene las siguientes extensiones aplicativas de la lista enlazada:

A. Listas enlazadas simples:

Una lista enlazada simple se define por:

- El tipo de sus elementos: el campo de información donde se almacenan los datos y el campo de enlace apunta al siguiente elemento lo define un puntero.
- Un puntero de cabecera que permite acceder al primer elemento de la lista.
- Un medio para detectar el último elemento de la lista: puntero nulo (NULL).

La representación gráfica de la lista enlazada simple se muestra de la siguiente manera:

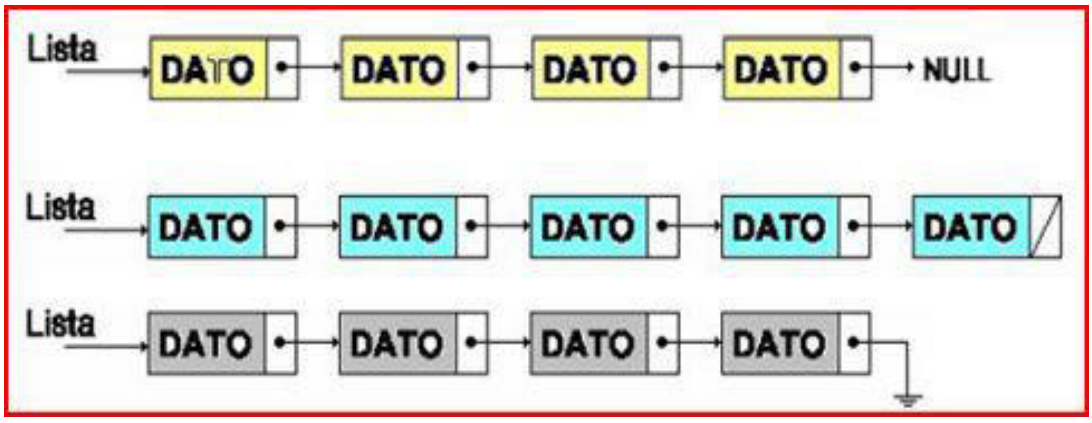

Figura 2.7: Representación gráfica de varios tipos de listas enlazadas simples referenciando NULL de varias maneras. [7]

B. Listas enlazadas dobles:

Las listas enlazadas dobles son un tipo de lista lineal en la que cada nodo tiene dos enlaces, uno que apunta al nodo siguiente, y el otro que apunta al nodo anterior.

Las listas doblemente enlazadas no necesitan un nodo específico para acceder a ellas, ya que presentan una gran ventaja comparada con las listas enlazadas y es que pueden recorrerse en ambos sentidos a partir de cualquier nodo de la lista, ya que siempre es posible desde cualquier nodo alcanzar cualquier otro nodo de la lista, hasta que se llega a uno de los extremos.

La representación gráfica de la lista enlazada doble se muestra de la siguiente manera:

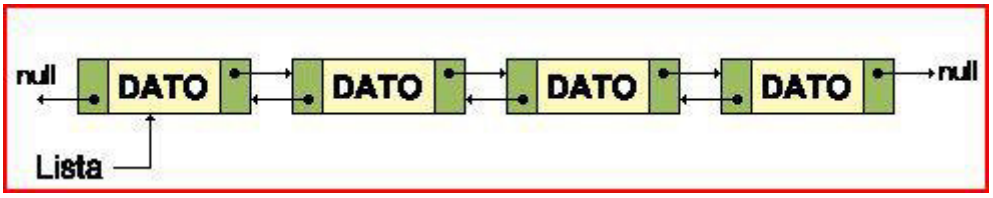

Figura 2.8: Representación gráfica de una lista enlazada doble. [8]

C. Listas enlazadas circulares:

Las listas circulares presentan las siguientes ventajas respecto de las listas enlazadas simples:

- Cada nodo de una lista circular es accesible desde cualquier otro nodo de ella. Es decir, dado un nodo se puede recorrer toda la lista completa. En una lista enlazada de forma simple sólo es posible recorrerla por completo si se parte de su primer nodo.
- Las operaciones de concatenación o unión y división de listas son más eficaces con listas circulares.

Pero también se presentan algunos inconvenientes:

- Se pueden producir lazos o bucles infinitos. Una forma de evitar estos bucles infinitos es disponer de un nodo especial que se encuentre permanentemente asociado a la existencia de la lista circular. Este nodo se denomina cabecera de la lista.

La representación gráfica de la lista enlazada circular se muestra de la siguiente manera:

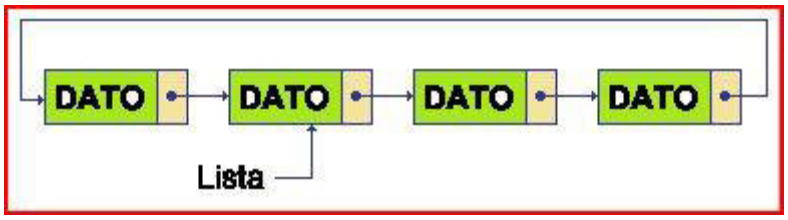

Figura 2.9: Representación gráfica de una lista enlazada circular. [9]

Las listas circulares tienen 2 tipos de variaciones principales que son las listas enlazadas circulares simples(un enlace por nodo) y listas enlazadas circulares dobles(dos enlaces por nodo).
La representación gráfica de ambos se muestra de la siguiente manera:

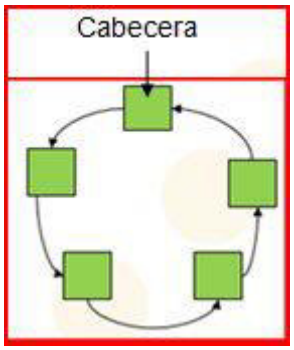

Figura 2.10: Representación gráfica de una lista enlazada circular simple. [9]

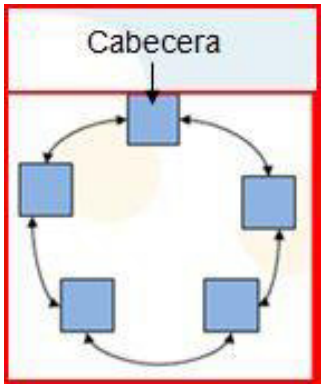

Figura 2.11 Representación gráfica de una lista enlazada circular doble. [9]

2.2 Operaciones principales de las listas enlazadas

Tenemos infinidad de operaciones como las operaciones de inserción, eliminación, búsqueda, entre otras como se muestra a continuación:

2.2.1 Creación del nodo

Crea al menos un nodo de referencia para reconocer la existencia de la lista. La representación gráfica de la creación de un nodo simple se muestra de la siguiente manera:

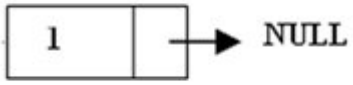

Figura 2.12: Representación gráfica de la existencia de un nodo en la lista. [10]

2.2.2 Recorrido de nodos

Recorre todos los nodos de la lista que apoya como base en la implementación de las demás funciones tales como: mostrar, insertar y eliminar nodo. La representación gráfica del recorrido de entrada y salida de los nodos se muestra de la siguiente manera:

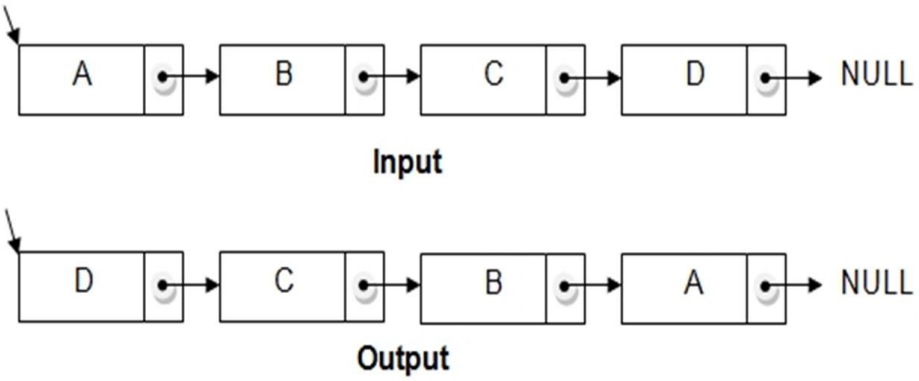

Figura 2.13: Representación gráfica del recorrido de entrada de la lista y como salida muestra el recorrido inverso de la lista. [10]

2.2.3 Insertar al inicio

Inserta como primer elemento un dato o registro de la lista de datos. La representación gráfica de la inserción al inicio de una lista enlazada se muestra de la siguiente manera:

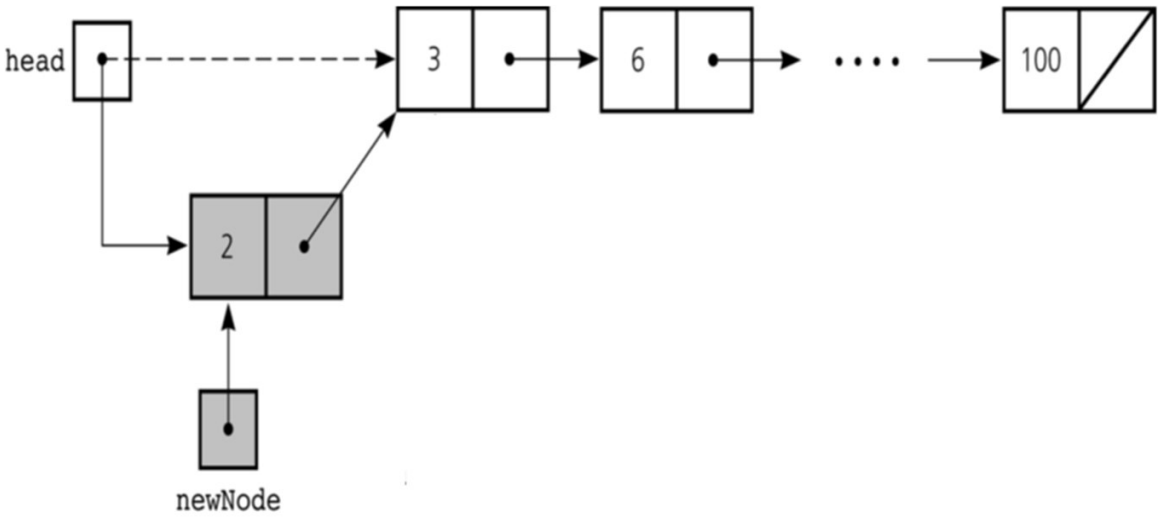

Figura 2.14: Representación gráfica de la inserción del primer elemento sombreado en la lista. [10]

2.2.4 Insertar al final

Inserta como último elemento un dato o registro de la lista de datos. La representación gráfica de la inserción al final de una lista enlazada se muestra de la siguiente manera:

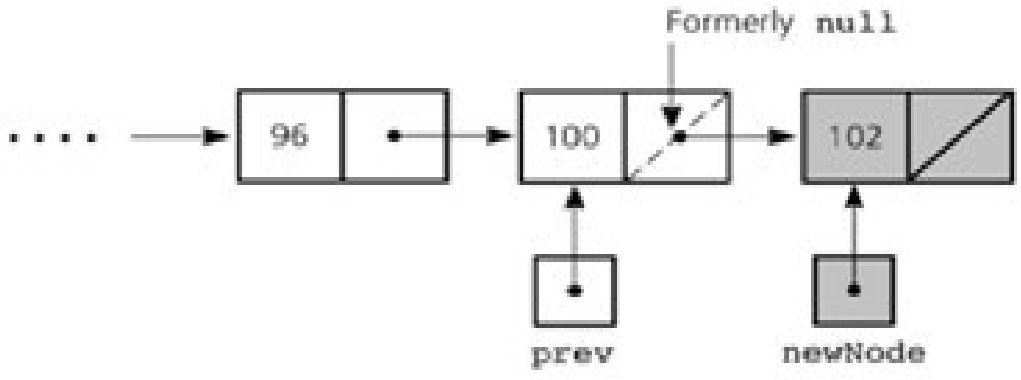

Figura 2.15: Representación gráfica de la inserción del último elemento sombreado en la lista. [10]

2.2.5 Insertar entre el inicio y el final

Inserta antes o después un dato o registro de la lista de datos. También, se inserta en una posición específica de la lista. La representación gráfica de la inserción entre el inicio y el final de una lista enlazada se muestra de la siguiente manera:

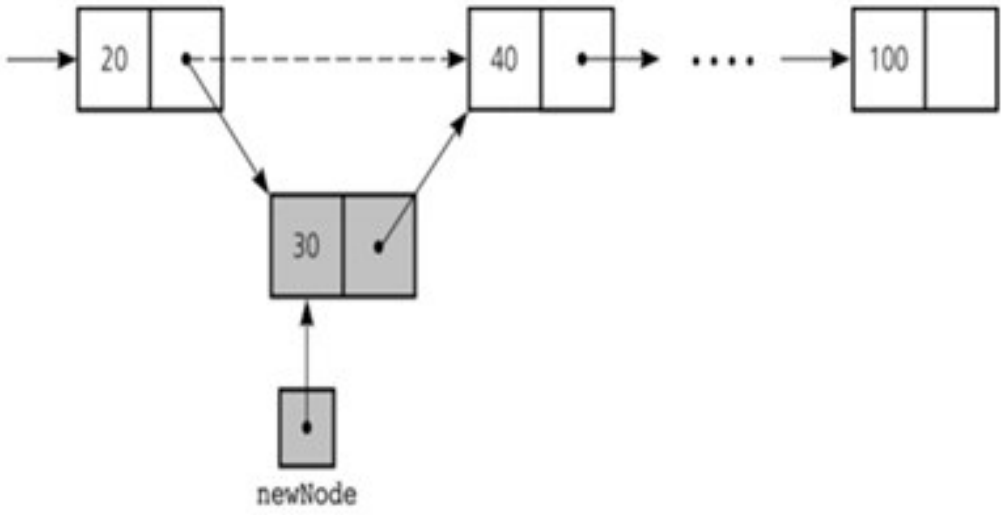

Figura 2.16: Representación gráfica de la inserción del elemento sombreado entre el inicio y final de la lista. [10]

2.2.6 Eliminar al inicio

Elimina como primer elemento un dato o registro de la lista de datos. La representación gráfica de la eliminación al inicio de una lista enlazada se muestra de la siguiente manera:

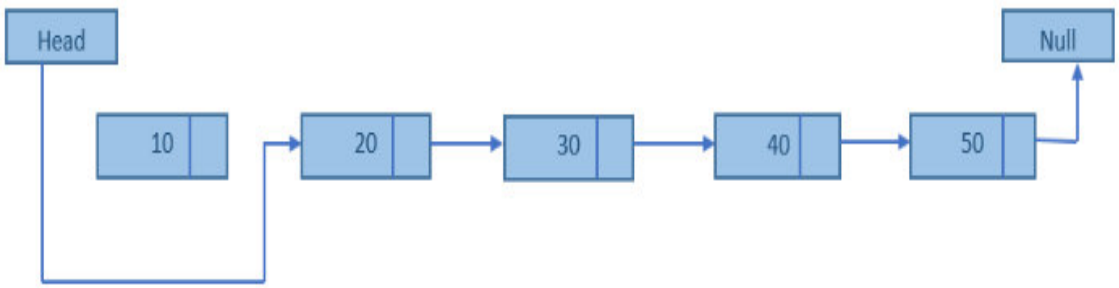

Figura 2.17: Representación gráfica de la eliminación del primer elemento en la lista. [11]

2.2.7 Eliminar al final

Elimina como último elemento un dato o registro de la lista de datos. La representación gráfica de la eliminación al final de una lista enlazada se muestra de la siguiente manera:

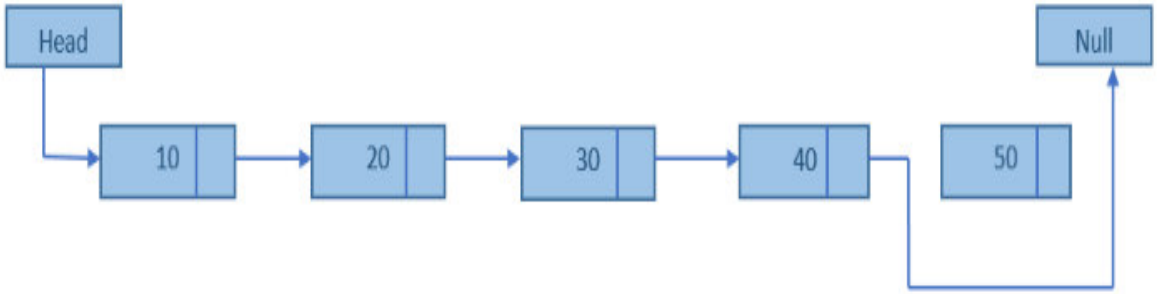

Figura 2.18: Representación gráfica de la eliminación del último elemento en la lista. [12]

2.2.8 Eliminar entre el inicio y el final

Elimina antes o después un dato o registro de la lista de datos. También, se elimina en una posición específica de la lista. La representación gráfica de la eliminación entre el inicio y final de una lista enlazada se muestra de la siguiente manera:

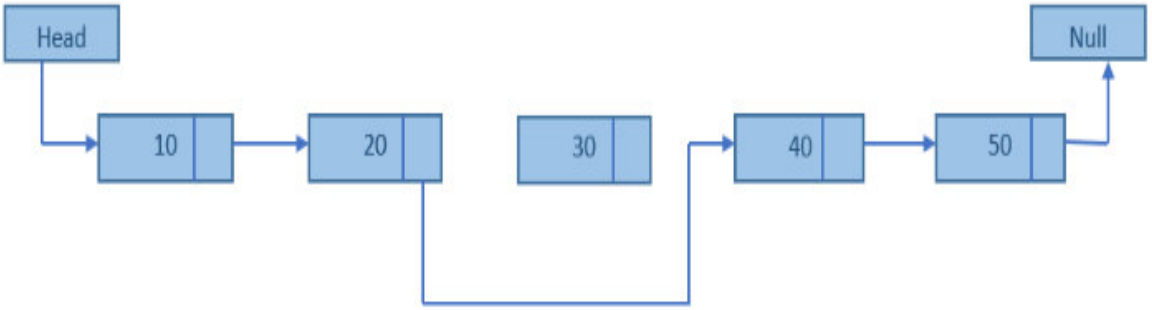

Figura 2.19: Representación gráfica de la eliminación de un elemento entre el inicio y el final de la lista. [13]

2.3 Limitaciones de las listas enlazadas

2.3.1 Ventajas

La lista dinámica posee las siguientes ventajas: es una estructura de datos dinámica, puede crecer y encogerse durante el tiempo de ejecución, las operaciones de inserción y eliminación son más fáciles de manipular, la utilización de memoria es eficiente, el tiempo de acceso es rápido, las estructuras de datos lineales como las pilas son fácilmente implementados.

En la Tabla 2.1, se muestran las principales ventajas y sus respectivas características de las listas enlazadas:

Ventajas	Características
La lista enlazada es dinámico por naturaleza.	- Al crear la estructura de la lista enlazada, la memoria será asignada en el tiempo de ejecución - En el tiempo de ejecución se puede asignar tanta memoria como sea posible. - Se puede asignar un número ilimitado de nodos, dependiendo del límite de asignación de memoria.
Las operaciones de inserción y eliminación son fáciles	- Las operaciones de inserción y eliminación de listas enlazadas son muy flexibles. - Se puede tanto insertar como eliminar para cualquier nodo en cualquier lugar fácilmente - No se tienen que cambiar nodos como la inserción de vectores. En la inserción de listas enlazadas, solo se tiene que actualizar los enlaces del nodo.
Utilización de memoria	- No se tiene que asignar memoria en el tiempo de ejecución. - La memoria es asignada durante el tiempo de ejecución y por requerimiento requerido, la estructura de datos de la lista enlazada proporciona un fuerte comando en la utilización de memoria.

Tabla 2.1. Principales ventajas de las listas enlazadas. [1]

2.3.2 Desventajas

En la tabla 2.2, se muestran las principales desventajas y sus respectivas características:

Desventajas	Características
Desperdicio de memoria	- El puntero requiere espacio adicional para el almacenamiento.
No hay acceso al azar	- En vectores, se puede acceder al n-simo elemento solo accediendo el índice del elemento [n]. - En las listas enlazadas no hay acceso aleatorio, así que se tiene que acceder a cada nodo secuencialmente.
Complejidad de tiempo	- El vector puede ser accedido aleatoriamente, mientras las listas enlazadas no pueden ser accedidas aleatoriamente. - Los nodos individuales no son almacenados en localizaciones de memoria contiguas. - El tiempo de acceso para un elemento individual es de complejidad lineal(O(n)), mientras el del vector es de complejidad de variable(O(1)).
El recorrido en reversa es dificultoso	- Si se usa listas enlazadas, entonces es muy dificultoso recorrer la lista desde el último elemento. - Si se usa lista doblemente enlazada, entonces es más fácil recorrer desde el final, pero se incrementa el almacenamiento de espacio por el puntero adicional implementado.
Restricción de espacio de almacenamiento dinámico	- Cuando la memoria es dinámicamente asignada, se utiliza memoria del almacenamiento dinámico. - La memoria es asignada a la lista enlazada en el tiempo de ejecución si y solo si hay disponibilidad de espacio en el almacenamiento dinámico. - Si hay insuficiente espacio en el almacenamiento dinámico, entonces no se creará memoria adicional.

Tabla 2.2 Principales desventajas de las listas enlazadas. [2]

2.3.3 Listas VS. Vectores

En la tabla 2.3, se muestran las principales funcionalidades en base a la comparación de las listas enlazadas y vectores, respecto a las ventajas y desventajas mencionadas anteriormente:

Funcionalidades	Comparación de Listas VS. Vectores
Acceso aleatorio y secuencial	- En los vectores, los elementos pueden ser accedidos usando variables suscritas como [0], [1]. [2], etc. - En las listas enlazadas se tiene que recorrer por toda la lista por acceder a elementos. La complejidad lineal(O(n)) es requerida para acceder a los elementos. - Generalmente, en las listas enlazadas, los elementos son accedidos secuencialmente.
Estructura de memoria	- En los vectores, los elementos son almacenados en localizaciones de memoria contiguas. - En los vectores, no es necesario almacenar el próximo elemento en la localización de memoria consecutiva. - En las listas enlazadas, el elemento es almacenado en cualquier localización disponible, pero el puntero a esa localización de memoria es almacenado en el nodo previo.
Inserción y eliminación	- Los elementos de los vectores son almacenados en localizaciones de memoria consecutivos, mientras al insertar elementos se tiene que crear espacio para la inserción. - En las listas enlazadas, crear espacio e insertar elementos requieren más tiempo. - En los vectores, se tiene que borrar el elemento de una localización dada y luego mover todos los elementos sucesivos por una posición. - En la lista enlazada, se tiene que cambiar el campo de dirección de puntero al borrar un elemento.
Asignación de memoria	- En los vectores, la memoria debería ser asignado en el tiempo de compilación.

	- En listas enlazadas, la memoria puede ser asignado en el tiempo de ejecución. - En los vectores se usa la asignación de memoria estática, mientras que en las listas enlazadas se usa la asignación de memoria dinámica. - Las funciones de asignación de memoria dinámica son: malloc, calloc, delete, etc.

Tabla 2.3 Principales funcionalidades entre las listas enlazadas y los vectores.

[3]

CAPÍTULO III. ESTADO DEL ARTE

3.1 Comparación entre vectores y listas

- *[R.K.Gosh]* propone los siguientes puntos sobre las operaciones de listas enlazadas: La localización dinámica es apropiado para construir listas, árboles y grafos, se usa arreglos para almacenar una específica colección de ítems de datos, si la colección de ítems de datos así mismo crece y se encoge, entonces usar una lista enlazada es lo más apropiado. Esto permite las siguientes características: inserción, eliminación y búsqueda. Pero, las capacidades de acceso al azar del arreglo es perdido.

- Mientras *[Madam Umi Kalsum Hassan],* dice que existen diferencias claras entre los vectores y las listas enlazadas. El tamaño del vector es un número fijado y dicho tamaño necesita ser especificado durante la declaración; a comparación del tamaño de la lista enlazada que crece y se contrae de las inserciones y eliminaciones, y el máximo tamaño depende del amontonamiento de los elementos. La capacidad de almacenamiento del vector es estático, quiere decir que la localización es asignado durante el tiempo de compilación; a comparación de la capacidad de almacenamiento de la lista dinámica que es dinámica, esto quiere decir que el nodo es localizado durante el tiempo de ejecución. El orden y almacenamiento del vector es almacenado consecutivamente; mientras que la lista dinámica es almacenado al azar. El acceso al elemento del vector son de dos formas: método de acceso aleatorio o directo, y especificar el índice o subíndice; mientras el acceso al elemento de la lista dinámica son también de dos formas diferentes: método de acceso secuencial y comienzo de atravesar desde el primer nodo de la lista por el puntero. La búsqueda del vector se distingue de dos formas: búsqueda binaria y búsqueda lineal; mientras que la lista enlazada es mediante la búsqueda lineal.

- Además, *[Frank M. Carrano],* marca dos diferencias claras entre un vector y una lista enlazada. El vector tiene un tamaño fijado y los datos deben ser cambiados durante las inserciones y eliminaciones. La lista enlazada es capaz de crecer en tamaño cuantas veces sea necesaria y no requiere cambiar los datos durante las inserciones y eliminaciones.

- Al igual que *[Yinzhi Cao],* propone que el vector requiere un máximo estimado del tamaño de la lista, eso genera espacio desperdiciado y siempre necesita rastrear, actualizar el tamaño. Las listas enlazadas comparadas a la implementación del vector, es que las listas necesitan la implementación del puntero usando tanto espacio como los elementos que se usan en la lista, por tanto ahorran espacio; pero requiere espacio adicional por los punteros de cada nodo, por tanto gastan espacio en la implementación de usar nodos.

3.2 Algoritmos de representación de un nodo

- *El nodo de [R.K.Gosh]* se representa en la siguiente estructura del nodo:

  struct node
  { int info;
  struct node *next;
  }

  Declarar el nodo de la siguiente manera: La lista enlazada es una colección de ítems de datos donde cada ítem es almacenado en una estructura(nodo). La estructura por nodo puede ser declarado como sigue: Una estructura de nombre "node" compuesto conteniendo los siguientes elementos: una variable de tipo entero de nombre "info" y una estructura con el mismo nombre "node" y declarando un puntero de nombre "next".

- *El nodo de [Jennifer Rexford]* se representa en la siguiente estructura:

```
struct Node {
   const char *key;
   int value;
   struct Node *next;
};

struct Table {
   struct Node *first;
};
```

Descripción en código de la estructura de datos del nodo.

La representación gráfica del nodo en base al algoritmo se muestra como:

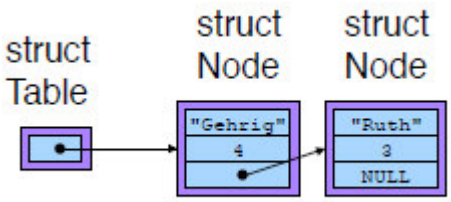

Figura 3.1: Descripción gráfica de la estructura de datos del nodo. [15]

Se crea la función "struct Node" declarando un registro de una constante de variable "key", un entero "value" y de nuevo llama a "struct Node" para apuntar al siguiente nodo. "struct Table" apunta al primer elemento del conjunto de elementos de la lista.

- *El nodo de [Madam Umi Kalsum Hassan]* se representa en la siguiente estructura:

  A. Sintaxis del nodo interno:

  ```
  struct nodename
  {       variable declarations;
          nodename *next;
  };
  ```

  Se crea "struct nodename" en la cual se almacenará el nodo, tendrá 2 componentes: "variable declarations" hace referencia de 1 tipo de dato y variable a múltiples tipo de datos y variable. Los múltiples tipo de datos y variable se le conoce como registro o estructura de datos. Por ejemplo puedo declarar "int Data" como único valor en mi información del nodo o declarar un registro de nota del curso de un alumno como "char nombre[20]", ID como "int ID", nota como "float nota", curso como "string curso". "nodename *next" hace referencia a que el puntero apuntará al siguiente y próximo nodo de la lista de elementos.

  B. Sintaxis del nodo externo:

  ```
  Nodename *pointer1,*pointer2,..,*pointerN
  ```

Los punteros externos son necesitados en una lista enlazada por los siguiente propósitos: mantener la lista por apuntar al primer nodo de la lista, atraviesa o visita los nodos de la lista, mantiene a los nuevos nodos y marca la localización del nodo a ser declarado o insertado.

En el algoritmo de la representación del nodo de *Madam Umi Kalsum Hassan,* se presenta una primera observación, en la sintaxis del nodo interno, clarificar que la información de los datos almacenados en cada nodo puede ser mediante un único valor o múltiples valores como es el caso de registro.
Una segunda observación, es que en la sintaxis del nodo externo, se especifica una cantidad de nodos referenciados en la que vamos a trabajar en la implementación de todas las operaciones de las listas tales como la inserción, eliminación, búsqueda, editar valores, guardar valores, etc.

C. Declaración de un nodo dinámico:

```
struct NumNode
{    int value;
   NumNode * Next;
};
NumNode *head, *current, *temp, *Nptr;
head=NULL; //the list is empty
```

Crea una estructura "struct NumNode" en donde se referenciará cada nodo. "NumNode *head" mantiene a la lista, "NumNode *current" atraviesa o visita la lista, "NumNode *temp" marca un nodo en la lista, "NumNode *Nptr" asigna un nuevo nodo.

- *El nodo de [Trupti Agrawal]* se representa en la siguiente estructura:

```
typedef struct list_node *list_pointer;
struct list_node {
        char data [4];
        list_pointer link;
};
```

Una estructura nodo que contiene al dato y la autoreferencia de la misma estructura del nodo especificado con el tipo de dato.
Considerar los siguientes puntos importantes en las operaciones de las listas: la creación del nodo cabeza de la lista que se representa como "list_pointer ptr =NULL", la asignación de memoria del nodo que se representa de la siguiente manera: "ptr=(list_pointer) malloc (sizeof(list_node))" y la liberación de espacio que se representa como "free(ptr)".

En el algoritmo de representación del nodo de *Trupti Agrawal*, se presenta una observación, al declarar la estructura del nodo lo hace con la función especial reservada "typedef" en la cual le da un nuevo nombre a la estructura del nodo. Con "typedef struct list_node *list_pointer", quiere decir que la estructura "struct list_node" se va a nombrar "list_pointer". Una segunda observación es que el autor establece y resalta la autoreferencia del mismo tipo de dato en la creación de estructura del nodo.

3.3 Algoritmos de creación de la lista

- *La creación de la lista de [R.K.Gosh]* se representa en la siguiente estructura:

```
Struct node *newNode{
        newNode=malloc(sizeof(struct node));
        (*newNode).info=0;
        // newNode->info=0;
}
```

Se declara la cabeza de la lista de la siguiente manera: La lista enlazada es accesado a su primer nodo. Los nodos subsecuentes pueden ser accedidos por usar el puntero "next". Después de declarar, una lista vacía es declarado a NULL o vacío. Crear una lista es hecho por crear nodos, uno después de otro.
Existen funciones especiales que ayudan a la creación de nodos y la separación de espacio de memoria para lista. Uno es: (*newNode.info) es semejante o equivalente a newNode->info, su función es establecer información para el dato. El otro es: newNode=malloc(sizeof(struct nodo)),

su función es cargar un espacio libre de la estructura nodo en bits de la memoria.

En el algoritmo de creación de la lista de *R.K.Gosh,* se presenta una primera observación, newNode->info pone valor entero de 0, para referenciar que existe algún valor númerico en la lista como precondición antes de empezar las operaciones de inserción y eliminación. Una segunda observación es que en vez de new struct node, se usa el común separador de espacio de nodos: malloc(sizeof(struct node)). Una tercera observación es que cuando separa el espacio en la memoria por cada nodo "newNode" llamado en la lista, él propone valores nulos o 0, cuando no es necesario detallar algún valor, cuando se ingresa otros valores numéricos.

- *La creación de la lista de [Jennifer Rexford]* se representa en la siguiente estructura:

  A. Crear lista:

```
struct Table *Table_create(void) {
   struct Table *t;
   t = (struct Table*)
      malloc(sizeof(struct Table));
   t->first = NULL;
   return t;
}
```

```
struct Table *t;
…
t = Table_create();
…
```

Descripción en código de la estructura de datos del nodo.

La representación gráfica de la creación de la lista en base al algoritmo, se muestra como:

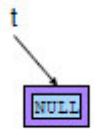

Figura 3.2: Descripción gráfica de la estructura de datos del nodo. [15]

En la función struct Table *Table_create() se referencia al puntero t, luego se le da un espacio de memoria por medio de malloc(sizeof()), después "t->first" es apuntado hacía el vacío de la nueva lista. Con todo este procedimiento se

reservó espacio de memoria para la lista desconocida y con la que se va a implementar las operaciones de inserción, búsqueda y eliminación.

En el algoritmo de creación de la lista de *Jennifer Rexford*, se presenta una observación, en vez de usar el tipo de dato "cons char", se pudo haber trabajado con el tipo de dato "string".

- *La creación de la lista de [Madam Umi Kalsum Hassan]* se representa en la siguiente estructura:

  A. Creación de un nodo dinámico:

  Nptr = new NumNode;
  cout<<"Enter a number: ";
  cin>>Nptr->value;
  Nptr->Next=NULL;

  A partir de "new NumNode" se asigna un espacio de memoria al nodo y se asigna a "Nptr". Luego, se procede a almacenar los datos: "Nptr->value" como valor entero, después "Nptr->Next=NULL" hace que el nodo con el elemento ingresado apunte a vacío o nulo.
  La representación gráfica de la creación del nodo dinámico en base al algoritmo, se muestra como:

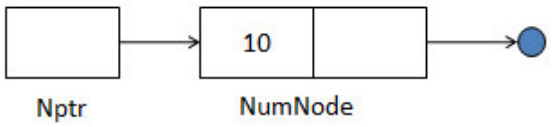

Figura 3.3: Descripción gráfica de la creación de un nodo dinámico. [16]

B. Creación de una lista enlazada:

Nptr = new NumNode
cout<<"Enter a number: ";
cin>>Nptr->value;
Nptr->Next=NULL;
if (head==NULL)
head=Nptr;

La representación gráfica de la creación de la lista en base al algoritmo, se muestra como:

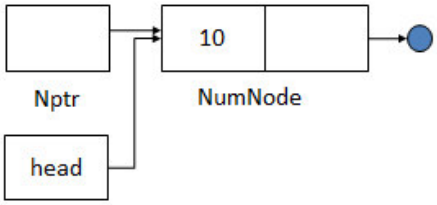

Figura 3.4: Descripción gráfica de la creación de la lista enlazada. [16]

A partir de "new NumNode" se asigna un espacio de memoria al nodo y se asigna a "Nptr". Luego, se procede a almacenar los datos: "Nptr->value" como valor entero, después "Nptr->Next=NULL" hace que el nodo con el elemento ingresado apunte a vacío o nulo. Si "head==NULL", quiere decir que la cabeza actualizada de la lista selecciona al primer elemento de la lista.

- *La creación de la lista de [Frank M. Carrano]* se representa en la siguiente estructura:
  Un nodo es dinámicamente asignado:
  Node *p;
  p=new Node;

  Se declara un Nuevo nodo arbitrario en "Node *p" y se le da un espacio de memoria a partir de "p=new Node".
  El puntero cabeza es el que hace referencia al primer nodo de la lista enlazada. Si la cabeza es nulo o vacío, la lista enlazada está vacía.
  La representación gráfica de la creación de la lista llena en base al algoritmo, se muestra como:

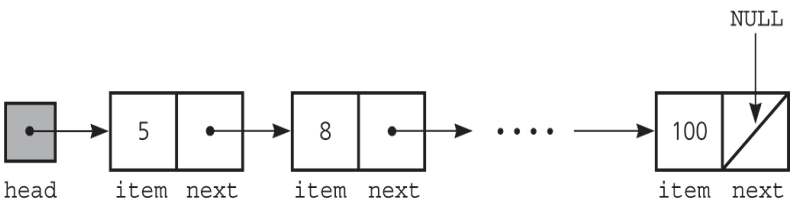

Figura 3.5: Descripción gráfica del puntero cabeza apuntando al primer elemento de la lista. [17]

La representación gráfica de la creación de la lista vacía en base al algoritmo, se muestra como:

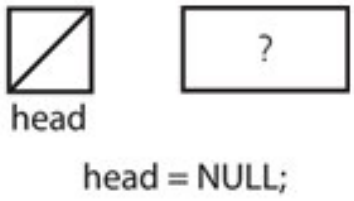

Figura 3.6: Descripción gráfica del puntero cabeza apuntando a nulo. [17]

- *La creación de la lista de [Trupti Agrawal]* se representa en la siguiente estructura:

  A. Crear dos nodos en una lista:

```
list_pointer create2( )
{
        list_pointer first, second;
        first = (list_pointer) malloc(sizeof(list_node));
        second = (list_pointer) malloc(sizeof(list_node));
        second -> link = NULL;
        second -> data = 20;
        first -> data = 10;
        first ->link = second;
        return first;
}
```

Se crean dos nodos “first” y “second” con “list_pointer first, second” y se les asigna un espacio de memoria a cada uno con “(list_pointer) malloc(sizeof(list_node))” . Se da valor 20 al nodo “second” y éste nodo apunta a nulo. Luego, se da valor 10 al nodo “first” y éste nodo apunta al nodo “second”.

La representación gráfica de la creación de la lista en base al algoritmo, se muestra como:

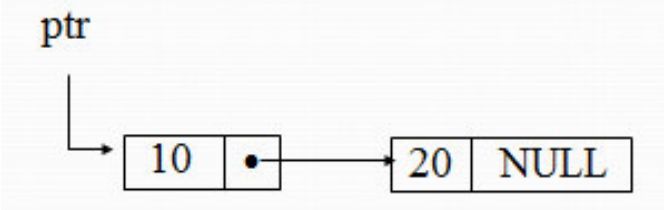

Figura 3.7: Descripción gráfica de la creación de los nodos con valores 20 y 10 con la cabeza nodo "ptr". [19]

En el algoritmo de creación de la lista de *Trupti Agrawal*, se presenta la primera observación, en la inserción de nuevo nodos se usa malloc(sizeof()) en vez de new struct(), y la eliminación de nodos se usa free() como referencia y no siendo necesario en la implementación. Una segunda observación es que en la función de crear nodos de la lista se hace de manera manual y no de manera automática llamándolos desde variables.

3.4 Algoritmos de búsqueda de elementos en la lista enlazada

- *La búsqueda de elementos en la lista enlazada de [R.K.Gosh]* se representa en la siguiente estructura:

  Con las siguientes condiciones: la lista podría ser una lista vacía y el elemento a ser insertado podría ya estar presente.

```
Struct node *searchList(struct node *list, int n){
        Struct node *p = list;
        while(p!=NULL){
        if(p->info == n)
                break;
        else
                p=p->next;
}
return p;
}
```

La función "Struct node *p = list", es referenciar el nodo "p" al inicio del conjunto de datos de la lista "lista". El bucle "while" va iterando hasta que el recorrido del nodo "p" referencie hasta el final de la lista "NULL". El nodo "p" referencia el valor conocido "n" de la lista, entonces finaliza la búsqueda

con “break”. En “p=p->next”, el nodo “p” va recorriendo a través de la lista de sus nodos siguientes, uno por uno. Finalmente, retorna el valor buscado.

En el algoritmo de búsqueda del nodo de *R.K.Gosh*, se presenta una observación, en vez de usar aplicaciones lógicas de listas, sale de manera abrupta del bucle “while” por medio de break.

- *La búsqueda de elementos en la lista enlazada de [Jennifer Rexford]* se representa en la siguiente estructura:

  A. Buscar en la lista

```
int Table_search(struct Table *t,
    const char *key, int *value) {
    struct Node *p;
    for (p = t->first; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}
```

```
struct Table *t;
int value;
int found;
…
found =
    Table_search(t, "Gehrig", &value);
…
```

Descripción en código de la búsqueda de datos en la lista.

La representación gráfica de la búsqueda de “Gehrig” en la lista en base al algoritmo, se muestra como:

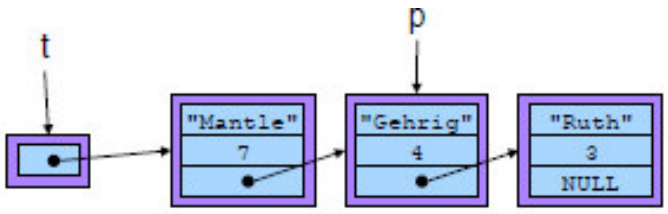

Figura 3.8: Descripción gráfica de la búsqueda de datos en la lista. [15]

La función “Table_search()” se encarga de buscar una cadena de caracteres que existe en la lista de caracteres. Si existe retorna el valor de 1, caso contrario retornará el valor de 0.

Primero referencia un nodo de apoyo denominado “p” y le da un espacio de memoria para poder trabajar en la inserción de la lista. Luego, hace una iteración por toda la lista desde “p=t->first hasta p=NULL, de tal manera que “p=p->next” avance de elemento a elemento de la lista. Dentro de esa

iteración se compara la cadena "p->key" con la cadena "key" que es la que queremos buscar. Dentro de la comparación de las llaves, se procede a almacenar el valor "p->value" de la lista

A la variable "*value" para verificar que el valor del elemento es de dicha lista. Si este proceso logra encontrar la cadena de caracteres deseado, entonces regresará el valor de 1, si no encuentra ningún valor que se quiere encontrar de la lista o la lista está vacía procederá a regresar el valor de 0.

Luego de referenciar "Struct Table *t" se selecciona los valores a buscar en "Table_search()", la cual referencia al puntero del inicio de la lista, el nombre de la cadena a buscar y el valor numérico a almacenar del valor de la lista que se quiere buscar. Luego, si encuentra el valor de la cadena que se quiere buscar, entonces almacena 1 en la variable "found".

En el algoritmo de búsqueda del nodo de *Jennifer Rexford*, se presenta una observación, en: "*value=p->value", no era necesario almacenar el valor en otra variable, el valor "p->value" pudo haberse retornado a la misma función "Table_search()".

- *La búsqueda de elementos en la lista enlazada de [Madam Umi Kalsum Hassan]* se representa en las siguientes estructuras:

  A. Visitar una lista enlazada:

  Se tienen dos propósitos:

  A. Procesar cada nodo en la lista: Por ejemplo, tenemos que visitar una lista para sumar todos los valores de la lista y calcular el valor promedio:

```
sum = 0;
count = 0;
current = head; // point the first node in the list
while (current != NULL)
{  sum = sum+current->data;
   count++; // count the node
   current = current->Next;
}
average = sum/count;
```

Inicializamos “sum=0” y “count=0” para almacenar valores sumados y contados de los valores de los nodos. “current=head” apunta al primer nodo de la lista, luego “while (current != NULL)” hará un bucle hasta que recorra toda la lista y el último nodo referencie a nulo. Dentro del bucle, “sum = sum+current->data” actualice el valor de sumado de todos los elementos de la lista uno por uno, “count++” suma la cantidad de nodos por cada iteración, “current = current->Next”, significa que el nodo “current” viaja por toda la lista hasta que llegue a nulo. Finalmente, cuando haya recorrido toda la lista, sale del bucle y comienza a hallar el promedio de las variables calculadas.

B. Encontrar el último nodo de la lista:

```
while (current->Next != NULL)
        current=current->Next;
```

En el bucle “while (current->Next != NULL)” se asegura que el nodo actual y referenciado desde el inicio de la lista viaje por toda la lista hasta que sea nulo o vacío, “current=current->Next” se asegura que visite cada nodo visite a otro nodo continuo y vecino de la lista.

- *La búsqueda de elementos en la lista enlazada de [Frank M. Carrano]* se representa en la siguiente estructura:

  A. Mostrar el contenido de una lista enlazada:

  Una operación de recorrido visita cada nodo de la lista enlazada. Un puntero “cur” mantiene el rastreo del actual nodo:

```
for (Node *cur = head; cur != NULL; cur = cur->next)
        cout << cur->item << endl;
```

  La lista referenciará y asignará espacio de memoria a “Node *cur”, luego recorrerá desde el puntero cabeza “head” hasta que el puntero sea nulo moviéndose de uno por uno a la “lista cur = cur->next”. Se mostrará los valores almacenados por cada nodo y uno por uno mediante “cout<<cur-item”.

  La representación gráfica de la búsqueda y recorrido de la lista en base al algoritmo, se muestra como:

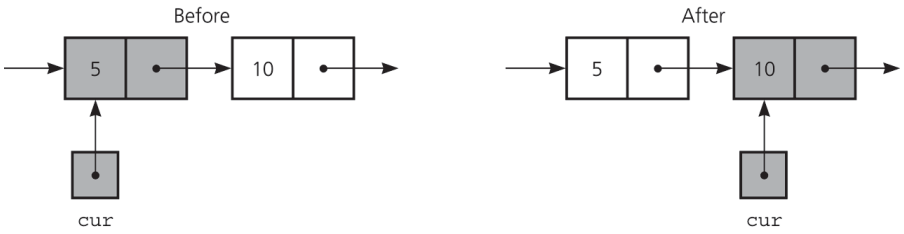

Figura 3.9: Descripción gráfica de cómo el nodo "cur" recorre la lista desde el inicio de la lista y seguido del segundo elemento de la lista. [17]

En el algoritmo de la búsqueda del nodo de *Frank M. Carrano*, se presenta una observación, en el muestreo del contenido de la lista, se declara todo el recorrido de la lista mediante un bucle "for". Más ordenado y eficiente sería el uso del bucle "while" y establecer orden en el recorrido de las variables o nodos en la lista.

- *La búsqueda de elementos en la lista enlazada de [Yinzhi Cao]* se representa en la siguiente estructura:

  Imprimir los elementos de una lista enlazada:

```
void DisplayList(Node* ptrToHead)
{
 int num              =      0;
Node* currNode        =      ptrToHead;
 while (currNode != NULL){
        cout << currNode->data << endl;
        currNode      =      currNode->next;
        num++;
 }
 cout << "Number of nodes in the list: " << num << endl;
}
```

El parámetro de la función al ingresar es el nodo cabeza "*ptrToHead" de la función de mostrar lista. Se inicializa "num=0" y en "Node* currNode=ptrToHead", el nodo referenciado "currNode" apunta al primer elemento elemento de la lista "ptrToHead". Mientras el bucle "while (currNode != NULL)" hace recorrer toda la lista con "currNode=currNode-

>next", en la cual se encarga de recorrer nodo por nodo, dentro del bucle se cuenta la cantidad de nodos de la lista con la variable "num". Una vez que sale del bucle al recorrer toda la lista, escribe un mensaje de la cantidad de nodo de la lista con la variable "num".

3.5 Algoritmos de inserción de elementos en la lista enlazada

- *La inserción de elementos en la lista enlazada de [R.K.Gosh]* se representa en las siguientes estructuras:

  A. Insertar al inicio de la lista:

  Se muestra el siguiente algoritmo de la inserción al inicio de la lista enlazada de un dato específico:

```
struct node *addToList(struct node *list, int n){
        struct node *newNode;
        if(searchList(list, n)!=NULL){
                Printf("%dvalor existe, inserción no es permitida", n);
                Return list;
        }
        Else{
                newNode = malloc(sizeof(struct node));
                if(newNode == NULL){
                        printf("creación de nodo fallido\n);
                else{
                        newNode->info=n;
                        newNode->next=list;
                }
        }
        Return newNode;
}
```

  La representación gráfica de la inserción al inicio de la lista vacía en base al algoritmo, se muestra como:

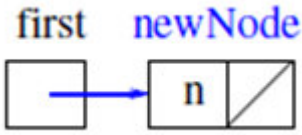

Figura 3.10: Descripción gráfica de inserción al inicio de una lista vacía. [14]

La representación gráfica de la inserción al inicio de la lista llena en base al algoritmo, se muestra como:

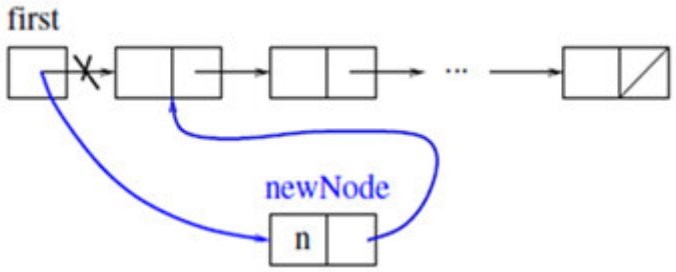

Figura 3.11: Descripción gráfica de inserción al inicio de una lista con elementos. [14]

La función "Struct node *p = list", es referenciar el nodo "p" al inicio del conjunto de datos de la lista "lista". Esta expresión: "searchList(list, n)!=NULL"; permite buscar el dato de valor "n" y compara si existe dicho dato o no. Si existe valor, imprime un mensaje de que la inserción no es permitida y regresa la lista actual. Si no existe valor, se crea nuevo espacio para un nuevo elemento de la lista permitiendo la inserción de un nuevo valor y apuntando al primer nodo de la lista. Finalmente, retorna el nodo creado con el valor especificado.

B. Insertar al final de la lista:

El autor propone los siguientes puntos: navegar la lista enlazada para alcanzar el último nodo y manejar el caso de la lista enlazada vacía. Se muestra el siguiente algoritmo de la inserción al inicio de la lista enlazada de un dato específico:

```
Struct node *endList(struct node *list){
        Struct node *p=list;
```

```
        if(P==NULL)
                return NULL;
        while(p->next!=NULL)
                p=p->next;
        Return p;
}
```

La función “Struct node *p = list”, es referenciar el nodo “p” al inicio del conjunto de datos de la lista “lista”. El nodo “p” referencia el valor nulo o no existente de la lista, entonces finaliza la búsqueda y retorna nulo. En “p=p->next”, el nodo “p” va recorriendo a través de la lista de sus nodos siguientes, uno por uno. Cuando lo encuentre, retorna el valor buscado.

Se muestra el siguiente algoritmo de la inserción al final de la lista enlazada de un dato específico:

```
Struct node *appendToList(struct node *list, int n){
        Struct node *newNode, *p;
        If(searchList(list, n)!=NULL)
                Printf(“%dexiste en la lista, anexo no permitido\n”, n);
        Else{
                newNode=malloc(sizeof(struct node));
                newNode->info=n;
                newNode->next=NULL;
                p=endList(list);
                if(p!=NULL)
                        p->next=newNode;
                else
                        list=NewNode;
        }
        Return list;
}
```

La representación gráfica de la inserción al final de la lista vacía en base al algoritmo, se muestra como:

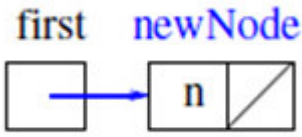

Figura 3.12: Descripción gráfica de inserción al final de una lista vacía. [14]

La representación gráfica de la inserción al final de la lista llena en base al algoritmo, se muestra como:

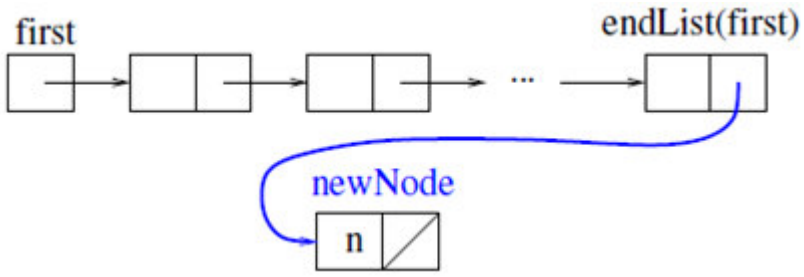

Figura 3.13: Descripción gráfica de inserción al final de la lista con elementos. [14]

La función "Struct node *p = list", es referenciar el nodo "p" al inicio del conjunto de datos de la lista "lista". Esta expresión; "searchList(list, n)!=NULL"; permite buscar el dato de valor "n" y compara si existe dicho dato o no. Si existe valor, imprime un mensaje de que la eliminación no es permitida. Si no existe valor, se crea nuevo espacio para un nuevo elemento de la lista permitiendo la inserción de un nuevo valor y apuntando al último nodo de la lista. Luego, "p=endList(list)" permite ir al final de la lista, luego comparar si tiene elementos o no tiene ningún elemento. Finalmente, retorna el nodo creado con el valor especificado.

En el algoritmo de inserción del nodo de *R.K.Gosh*, se presenta una observación, referenciar la inserción de elementos de la listas a partir de la función searchList(list, n), asegurando que no haya elementos repetidos.
Una segunda observación es que a pesar que ha dado un espacio de memoria al "newNode", se verifica si "newNode" está vacío en espacio de memoria.

- *La inserción de elementos en la lista enlazada de [Jennifer Rexford]* se representan en la siguiente estructura:

  A. Agregar al inicio de la lista:

```
void Table_add(struct Table *t,
    const char *key, int value) {
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    p->key = key;
    p->value = value;
    p->next = t->first;
    t->first = p;
}
```

Descripción en código de la inserción al inicio de la lista.

La representación gráfica de la inserción al inicio de la lista con 3 conjuntos de valores en base al algoritmo, se muestra como:

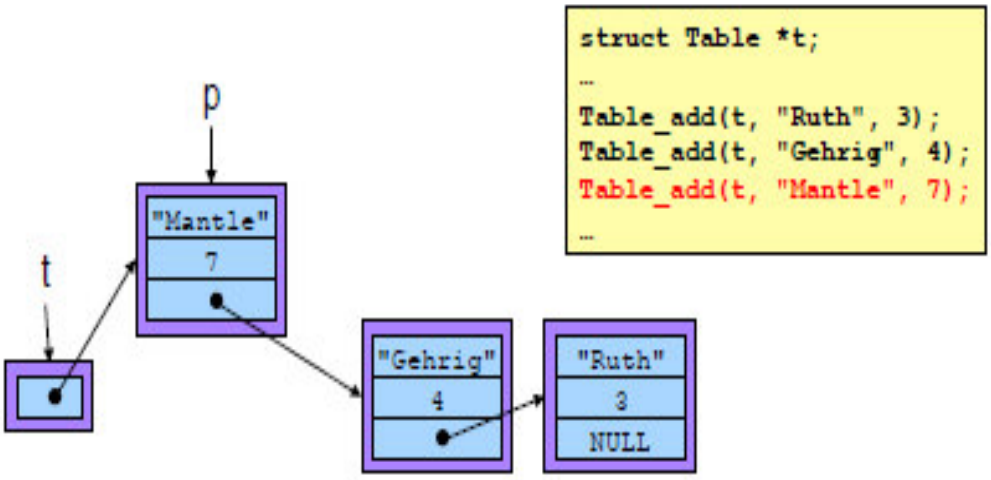

Figura 3.14: Descripción gráfica de la inserción al inicio de la lista. [15]

La función "Table_Add()" se encarga de insertar el valor entero y la cadena de caracteres al inicio de la lista de datos.

Primero referencia un nodo de apoyo denominado "p" y le da un espacio de memoria para poder trabajar en la inserción de la lista. "p->key" referencia a la cadena de caracteres, "p->value" referencia al valor númerico, "p->next" referencia al "t->first", dando a entender que el nodo que nosotros queremos insertar se ubica en una posición atrás del primer elemento actual de la lista.

Finalmente, “t->first=p”, la cabeza t que estaba ubicada como referencia al primer elemento de la lista anterior, ahora está actualizada y apuntando al elemento insertado con los valores almacenados.

Luego de referenciar “Struct Table *t”, se selecciona los valores a insertar en “Table_add()”, la cual referencia al puntero del inicio de la lista, el nombre de la cadena y el valor numérico a almacenar a la lista.

- *La inserción de elementos en la lista enlazada de [Frank M. Carrano]* se representa en las siguientes estructuras:

  A. Insertar un nodo entre dos nodos:

  newPtr->next = cur;

  prev->next = newPtr;

  El nuevo nodo a insertar es “newPtr”. Los dos nodos de apoyo para trabajar con la inserción del nuevo nodo son el nodo “prev” y el nodo “cur”, las cuales estarán juntas y luego separadas por el ingreso del nodo “newPtr”. “newPtr->next = cur”, quiere decir que el nuevo nodo “newPtr” apuntará al nodo “cur”. “prev->next = newPtr”, quiere decir que el nodo siguiente del nodo “prev” apuntará al nuevo “newPtr”.

  La representación gráfica de la inserción del valor 30 en la lista llena en base al algoritmo, se muestra como:

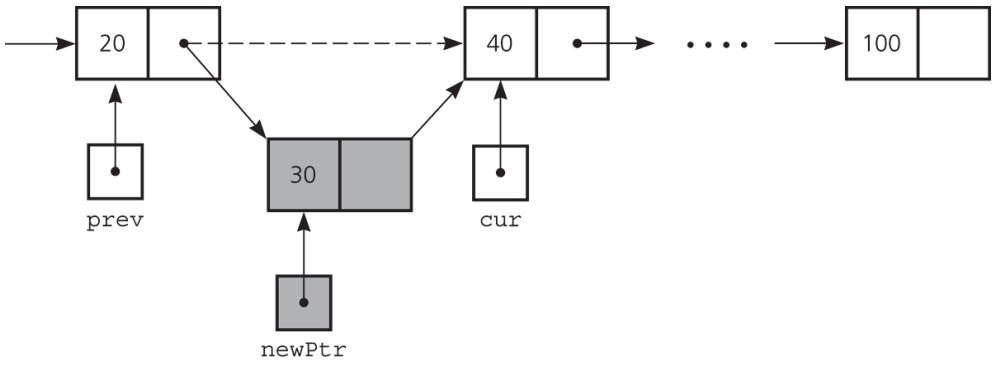

Figura 3.15: Descripción gráfica de la inserción del nodo “newPtr” con valor numérico de 30 junto y entre dos nodos con valor numérico 20 y 40 respectivamente. [17]

B. Insertar el nodo al comienzo:

newPtr->next = head;

head = newPtr;

El nuevo nodo a insertar es “newPtr”. “newPtr->next = head”, quiere decir que el nuevo nodo “newPtr” apuntará al nodo cabeza de la lista, el nodo “head”. “head = newPtr”, quiere decir que el nodo cabeza de la lista, el nodo “head”, apuntará al nuevo nodo “newPtr”.

La representación gráfica de la inserción del valor 2 al inicio de la lista llena en base al algoritmo, se muestra como:

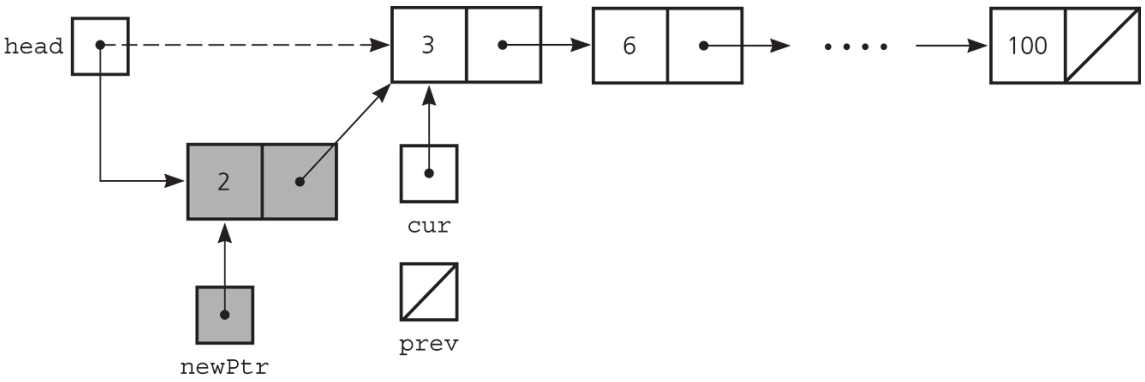

Figura 3.16: Descripción gráfica de inserción al inicio de la lista. [17]

En el algoritmo de inserción del nodo de *Frank M. Carrano*, se presenta una observación, en la inserción de un nodo entre dos nodos se trabaja con dos nodos auxiliares: “prev” y “cur”. También se puede trabajar con un solo nodo auxiliar como “prev” y en vez del otro nodo “cur” se le puede reemplazar por el nodo siguiente que apunta “prev”: “prev->next”.

- *La inserción de elementos en la lista enlazada de [Yinzhi Cao]* se representa en la siguiente estructura:

  A. Insertar un nuevo nodo con índice deseado:

```
Node* InsertNode(int index, double x) {
        if (index < 0) return NULL;
        int currIndex  =       1;
        Node* currNode       =       head;
        while (currNode && index > currIndex) {
```

```
            currNode    =    currNode->next;
            currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode    =    new    Node;
        newNode->data    =    x;
        if (index == 0) {
            newNode->next    =    head;
            head             =    newNode;
        }
        else {
            newNode->next    =    currNode->next;
            currNode->next   =    newNode;
        }
        return newNode;
}
```

Se recibe como parámetros el índice "index" y el tipo de dato a insertar "x" del nodo a la función de insertar nodo. Si el índice "index" es menor que 0, entonces regresará nulo o vacío. Se inicializa "currIndex" en 1, el nodo "currNode" apunta al primer elemento de la lista "head". El bucle "while (currNode && index > currIndex)", quiere decir que mientras exista el nodo "currNode" y no apunte a nulo, y el índice "Index" sea mayor a "currIndex" que es el número de nodos referenciado actuales que viajan para referenciar al nodo con la posición que deseamos ubicar con "currNode=currNode->next". El tipo de dato "currIndex" se va a actualizando con "currIndex++".

Se crea un nuevo nodo, se le asigna un espacio de memoria con "Node* newNode=new Node" y el tipo de dato "x" con "newNode->data=x". Si el valor del índice del dato es 0 con "if (index == 0)" se procede a ingresar el nuevo nodo antes del primer elemento a partir de "newNode->next=head ", luego la cabeza de la lista apuntará al nuevo nodo con "head =newNode ". Si el valor del índice es mayor a 0 con "else" se procede a que el nuevo nodo apunte a lo que apunte "currNode" con "newNode->next=currNode-

>next", luego, el nodo que apunte "currNode" apunta al nuevo nodo "NewNode con "currNode->next=newNode".

La representación gráfica de la inserción al inicio de la lista vacía y lista llena en base al algoritmo, se muestra como:

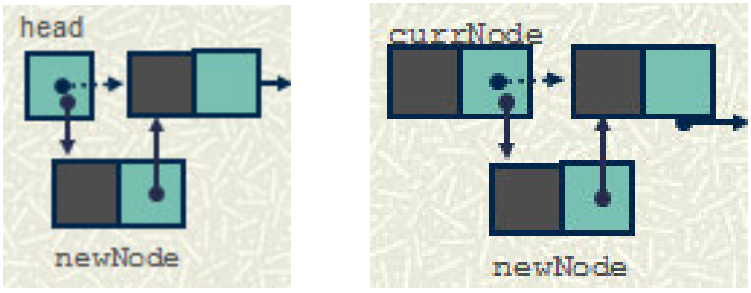

Figura 3.17 Descripción gráfica de inserción de nodo en una lista vacía y una lista con elementos. [18]

En el algoritmo de inserción del nodo de *Yinzhi Cao*, se hace una observación, en la función de insertar un nodo, se ha desarrollado por el método de saber en qué posición exacta se quiere insertar el nodo. El algoritmo está apto para desarrollarse sin falla al momento de insertar el índice del nodo si está entre insertar en una lista vacía o que el índice del nodo a insertar sea menor o igual a la cantidad de nodos de la lista actual. Y si el índice es menor de 0 o el índice del nodo a insertar es mayor a la cantidad de nodos de la lista actual, regresará un valor nulo.

- *La inserción de elementos en la lista enlazada de [Trupti Agrawal]* se representa en la siguiente estructura:

A. Inserción de un nodo en una lista:

```
void insert(list_pointer *ptr, List_pointer node)
{
        list_pointer temp;
        temp=(list_pointer)malloc(sizeof(list_node));
        if(IS_FULL(temp)){
                fprintf(stderr, "The memory is full\n");
                exit(1);
        }
        temp->data=50;
```

```
        if(*ptr){          //nonempty list
                temp->link = node->link;
                node->link = temp;
        }
        else{    //empty list
                temp->link = NULL;
                *ptr = temp;
        }
}
```

Los parámetros ingresantes de la función de ingresar nodos es el nodo cabeza "*ptr" y al salir es el nodo nuevo "node". Declarar un nodo "temp" referenciado con "list_pointer temp" y darle un espacio de memoria con "temp=(list_pointer)malloc(sizeof(list_node))". Si no hay memoria con "if(IS_FULL(temp))", entonces dará un mensaje que la memoria está llena. El nodo "temp" lo almacenamos con un valor de 50 con "temp->data=50". Si existe la cabeza como nodo "ptr" entonces el nodo "temp" apuntará al siguiente nodo del nodo "node" con "temp->link = node->link". El nodo "node" apuntará al nodo nuevo "temp" con "node->link = temp". Si no hay nodo cabeza, quiere decir que no existe lista, entonces el nodo nuevo "temp" apunta a nulo o vacío con "temp->link = NULL" y la cabeza del puntero "ptr" apunta a nulo o vacío con "*ptr = temp".

La representación gráfica de la inserción del valor 50 en la lista llena en base al algoritmo, se muestra como:

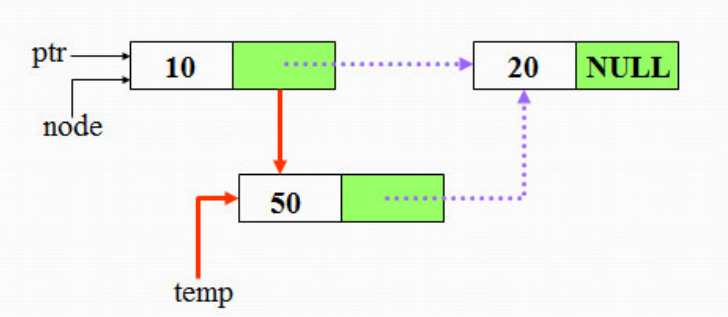

Figura 3.18: Descripción gráfica de la inserción del nodo con valor 50. [19]

3.6 Algoritmos de Eliminación de elementos en la lista enlazada

- *La eliminación de elementos en la lista enlazada de [Jennifer Rexford]* se representa en la siguiente estructura:

  A. Eliminar al inicio y uno por uno en la lista:

```
void Table_free(struct Table *t) {
   struct Node *p;
   struct Node *nextp;
   for (p = t->first; p != NULL; p = nextp) {
      nextp = p->next;
      free(p);
   }
   free(t);
}
```

```
struct Table *t;
…
Table_free(t);
…
```

Descripción en código de la eliminación al inicio y uno por uno de la lista.

La representación gráfica de la eliminación al inicio y hasta dejarlo nulo, en base al algoritmo, se muestra como:

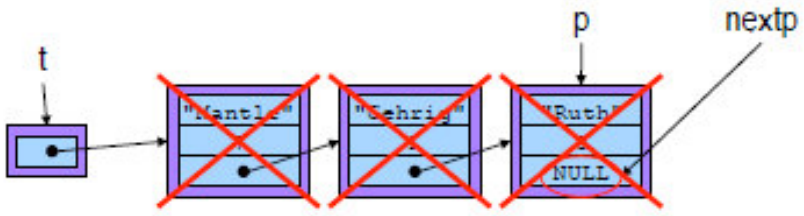

Figura 3.19: Descripción gráfica de la eliminación al inicio y uno por uno de la lista. [15]

La función "Table_free()" se encarga de eliminar al inicio elemento por elemento que posean el valor entero y la cadena de caracteres de la lista de datos.

Primero referencia un nodo de apoyo denominado "p" y "nextp", seguidamente les da un espacio de memoria para poder trabajar en la eliminación de la lista. Luego, hace una iteración por toda la lista desde "p=t->first" hasta "p=NULL", de tal manera que "p=p->next" avance de elemento a elemento de la lista. Dentro de esa iteración nextp apunta a la dirección del siguiente y adyacente nodo "p->next", luego, libera o borra la información del primer elemento de la lista "p", que contenía el primer valor entero y cadena de caracteres. La iteración va a seguir hasta que los elementos d ela

lista estén vacías, finalmente procederá a borrar el apuntador "t" que esta apuntando como un referente al inicio y sin elementos de la lista. Está función se encarga de eliminar uno por uno por el método de eliminar inicio. Luego de referenciar "Struct Table *t", se llama a la función "Table_free()".

En el algoritmo de eliminación del nodo de *Jennifer Rexford*, se presenta una observación, en vez de usar free(t) para liberar al nodo de la lista se pudo haber usar delete(t). En todo caso, no es necesario free() o delete(), porque los datos al desligarse de la lista, y luego recompilar el código fuente desde el inicio, los datos desligados de la lista se borran automáticamente de la memoria de la lista, por ende solo se gasta la memoria de la lista actual.

- *La eliminación de elementos en la lista enlazada de [Madam Umi Kalsum Hassan]* se representan en la siguientes estructuras:
  A. Eliminación de un nodo dinámico:
  delete Nptr;
  Nptr=Null;

  A partir de "delete Nptr" se borra el nodo apuntado por Nptr y se direcciona a NULL. Si la lista posee más de un nodo, entonces habrá necesidad de cambiar el puntero del registro antes que el primero sea borrado.
  La representación gráfica de la eliminación del valor 10 de la lista llena en base al algoritmo, se muestra como:

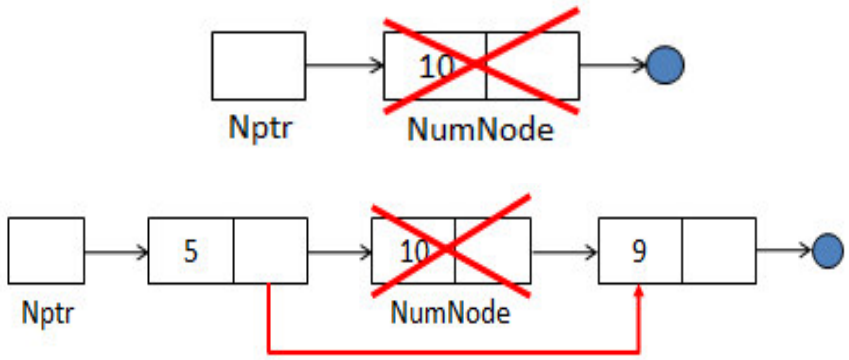

Figura 3.20: Descripción gráfica de la eliminación de nodos sea con un elemento o varios elementos. [16]

- *La eliminación de elementos en la lista enlazada de [Frank M. Carrano]* se representa en las siguientes estructuras:

A. Borrar un nodo interior:

prev->next=cur->next;

Se debe apoyar de dos nodos adicionales para trabajar con la lista: el nodo "prev" y el nodo "cur" en la cual estarán nuestros puntos estratégicos de inserción del nuevo nodo en el medio de ambos nodos. "prev->next=cur->next" quiere decir que el nodo "prev" está apuntando al nodo siguiente del nodo "cur", desplazando a "cur" fuera de la lista principal.

La representación gráfica de la eliminación del valor 8 de la lista llena en base al algoritmo, se muestra como:

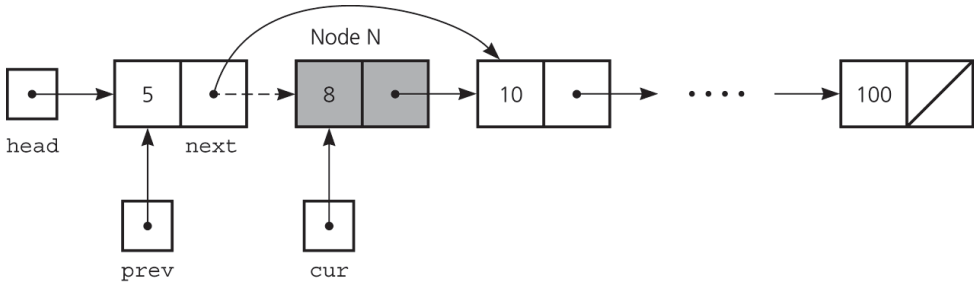

Figura 3.21: Descripción gráfica de la eliminación de un nodo interior con valor numérico 8 de la lista. [16]

B. Borrar el primer nodo:

head=head->next;

El nodo "head" está apuntando al primer elemento de la lista. "head=head->next" apunta al segundo elemento de la lista, desplazando al primer elemento de la lista fuera de la lista principal.

La representación gráfica de la eliminación al inicio y del valor 5 de la lista llena en base al algoritmo, se muestra como:

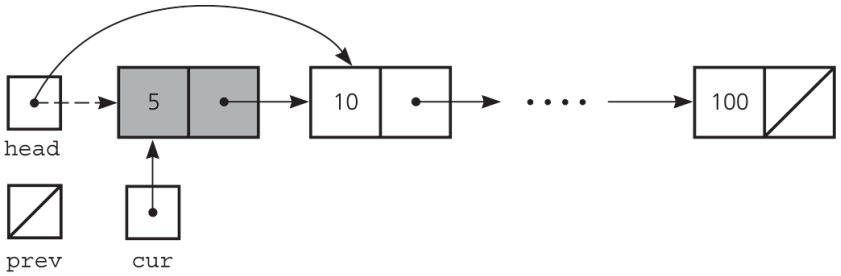

Figura 3.22: Descripción gráfica de la eliminación del primer nodo de la lista. [16]

C. Regresar el nodo borrado al sistema:

```
cur->next = NULL;
delete cur;
cur=NULL;
```

El nodo "cur" es el que se quiere borrar y se apunta a nulo, luego "delete cur" borra el nodo liberando espacio de memoria.

En el algoritmo de eliminación del nodo de *Frank M. Carrano*, se presenta una observación, en el regreso del nodo borrado del sistema y la lista principal, no es necesario hacer una función de eliminación del nodo de la memoria, porque al recompilar el código fuente ya desaparecerá el nodo desligado de la lista principal.

- *La eliminación de elementos en la lista enlazada de [Yinzhi Cao]* se representa en la siguiente estructura:

A. Borrar un nodo con dato deseado:

```
int DeleteNode(double x) {
        Node* prevNode      =       NULL;
        Node* currNode      =       head;
        int currIndex   =       1;
        while (currNode && currNode->data != x) {
                prevNode      =       currNode;
                currNode      =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next      =       currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
```

```
            }
            return currIndex;
        }
        return 0;
}
```

Se recibe como parámetros el tipo de dato a eliminar “x” de la función de eliminar nodo. Se declara 2 nodos de referencia, un nodo previo con “Node* prevNode” y un nodo actual con “Node* currNode”. Con while(), ubicó la posición del nodo previo “prevNode” con “prevNode=currNode”, y la ubicación del nodo actual “currNode”, con “currNode=currNode->next”; mientras currIndex va actualizándose, el bucle termina cuando siga existiendo “currNode” y no apunte a nulo o vacío de la lista principal, y a la vez que encuentra el dato referenciado “x” con “currNode->data != x”.

Si existe el nodo “currNode” con “if (currNode)”, entonces se abren 2 caminos: si existe “prevNode” con “if (prevNode)” entonces el nodo “prevNode” apunta a lo que apunta currNode con “prevNode->next=currNode->next”; luego procede a borrar “currNode” con “delete currNode”. El otro camino con “else” es que el nodo cabeza “head” apunta a lo que apunta el nodo “currNode con “head=currNode->next”; luego procede a borrar “currNode” con “delete currNode”.

Seguidamente de los 2 caminos, se procede a regresar el índice actual del nodo actual con “return currIndex”. Si no existe el nodo “currNode”, regresa valor nulo o 0.

La representación gráfica de la eliminación entre el inicio y final de la lista, y al inicio de la lista en base al algoritmo, se muestra como:

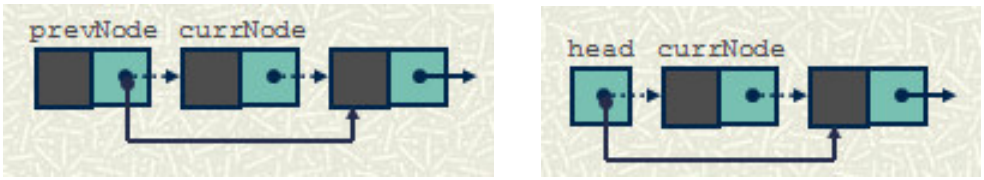

Figura 3.23: Descripción gráfica de la eliminación del nodo al inicio de la lista y al medio/final de la lista. [18]

En el algoritmo de eliminación del nodo de *Yinzhi Cao*, se hace una observación, en la función de eliminar un nodo, se ha desarrollado por el método de cuál es el valor a eliminar. Al momento de eliminar un nodo, se libera el espacio almacenado por dicho nodo. El algoritmo está apto para desarrollarse sin falla al momento de querer buscar el dato existente de la lista. Si no existe, retornará el valor de cero como mensaje de que no se encontró el dato deseado.

Al momento de eliminar un nodo, se libera memoria en el nodo que tiene el dato deseado a eliminar. No es necesario liberar espacio de memoria.

- *La eliminación de elementos en la lista enlazada de [Trupti Agrawal]* se representa en la siguiente estructura:

  A. Eliminación de un nodo en una lista:

```
void delete(list_pointer *ptr, list_pointer trail, list_pointer node)
{
        if(trail)
                trail->link = node->link;
        else
                *ptr = (*ptr)->link;
        free(node);
}
```

El parámetro entrante de la función de borrar nodo es el nodo cabeza "*ptr". El nodo de referencia "trail", con parámetro entrante "list_pointer trail", es el nodo precedente del nodo "node", con parámetro entrante "list_pointer node" que se quiere eliminar. Si existe el nodo "trail", entonces el nodo "trail" apuntará al siguiente nodo que apunta el nodo "node" que se quiere eliminar con "trail->link = node->link". Si no existe el nodo "trail" es porque la lista está vacía, entonces la cabeza del nodo "*ptr" apunta al siguiente nodo de "*ptr" con "*ptr = (*ptr)->link", actualizando el valor de la lista. Luego, se libera el nodo que se quiere borrar con "free(node)".

La representación gráfica de la eliminación del valor 50 de la lista llena en base al algoritmo, se muestra como:

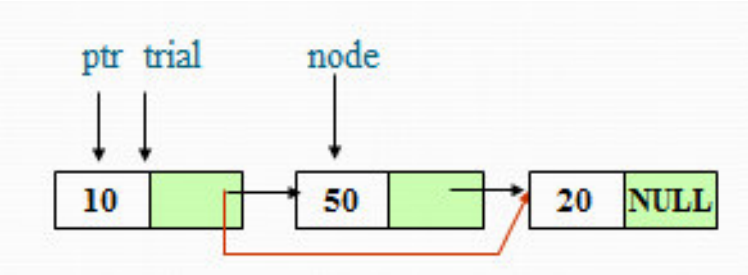

Figura 3.24: Descripción gráfica del nodo con valor 50. [19]

CAPÍTULO IV. APORTE

4.1 Vista General de la Implementación Base de las Listas Simples Estáticas:

La implementación del código se realizará en el lenguaje C++, se probará el funcionamiento de las diversas operaciones de las listas tanto dinámicas como estáticas.

La base de las operaciones es trabajar con un tipo de datos sea números o índices solicitados como entrada del usuario; el registro ampliado sea múltiples tipos de datos, se puede modificar en la reestructuración de dicho código internamente.

4.1.1 Vista panorámica del menú de las listas simples dinámicas:

El procedimiento de las funciones a realizar, se trabajarán en un nodo con un dato y un nodo con un enlace siguiente apuntando a otro dato. (Ver Anexo 4.1.1)

4.1.2 Vista panorámica del menú de las listas simples estáticas:

El procedimiento de las funciones a realizar se tomará como base la lógica del trabajo de nodos dinámicos de las listas simples dinámicas, y lo llamaremos nodos estáticos en la implementación de vectores y registros. (Ver Anexo 4.1.2)

En los siguientes puntos se tratará con más detalle la explicación del funcionamiento interno de dichos códigos, sus extensiones respectivas y pruebas con una específica cantidad de datos con la que trabaja cada función.

4.2 Operaciones con propias reglas lógicas tanto para listas dinámicas como listas estáticas.

Usar implementación lógica y técnicas base tomando como prioridad el recorrido de la lista de nodos y su posición específica para cada una de las siguientes funciones como se muestra en la Tabla 4.1:

Operación	Suboperaciones
Inicialización o creación de la lista	Reservar memoria para el primer nodo.
Insertar elementos en la lista	Al principio, al final.
	Antes o después de una posición específica.
	Antes o después de un elemento específico.
Eliminar elementos en la lista	Al principio, al final.
	En una posición específica.
	Con un dato específico.
Buscar elementos en la lista	En una posición específica.
	Con un dato específico.
Modificar elementos en la lista	En una posición específica.
	Con un dato específico.
Recorrer los nodos de la lista	Mostrar los datos visualmente.
	Elaborar técnicas de recorrido.
Comprobar si la lista está vacía	Verificar la existencia de nodos disponibles.
Cantidad de elementos de una lista	Verificar la cantidad de nodos disponibles.
Ordenar elementos en una lista	Ascendentemente, descendentemente.
Trabajar con archivos externos	Guardar y recuperar la lista de datos.

Tabla 4.1: Tabla de operaciones de la lista enlazada.

La descripción de la Tabla 4.1 se especifica por las siguientes operaciones y suboperaciones más resaltantes, tanto para listas dinámicas como las listas estáticas:

- **Crear las estructuras y nuevos nodos de la lista estática:**
  El objetivo de esta operación es crear estructuras de dato y la forma del nodo siguiente de la estructura del nodo y nuevos nodos sucesivos a partir de la creación de nodos en la lista.

Cuando se crea estructuras del nodo se ayuda de las posiciones o índices del vector.

Estas operaciones se asemejan al proceso de operaciones de listas dinámicas con el operador "new struct nodo" que reserva el espacio de memoria por cada nodo declarado.

A.1 Creación y nuevo nodo en listas dinámicas:

```
void crearlista(nodo **lista){
  *lista=NULL;
}
```

En esta función la lista se declara nulo desde el principio cuando no posee elementos.

```
p=new struct nodo;
```

Esta operación sirve para dar espacio dinámica a cada nodo y su respectivo dato o datos según el tipo de registro declarado.

A.2 Creación y nuevo nodo en listas estáticas:

```
void crear(nodo lista[max], int *cab, int *cab1){
  int i;
  *cab=-1;
  *cab1 =0;
  lista[0].sgte =1;
  for(i=1;i<max-1;i++)
  {
    lista[i].dato=0;
    lista[i].sgte = i+1;
  }
  lista[max-1].sgte =-1;
}
```

La cabeza de la lista “cab” es -1 referenciando a NULL, se agrega una cabeza auxiliar de la lista “cab1” que servirá como referencia o ayuda para la creación de cada nodo estático.

Dentro del bucle “for” se crea todas las estructuras posibles con el límite máximo declarado, simulando la creación de espacio de nodos estáticos. El sucesor siguiente del último elemento almacenado será -1 referenciando a NULL.

```
void nuevonodo(int *p, nodo lista[max], int *cab1)
{
  *p = *cab1;
  if(*cab1 == -1)
  {
    printf(" No hay suficiente memoria\n\n");
    exit(1);
  }
  *cab1 = lista[*cab1].sgte;
}
```

Al momento que se crea un nodo estático “p”, la cabeza auxiliar de la lista “*cab1” referenciará valor por valor incremental al nodo estático “p”.

Crea un nuevo nodo a partir de la posicion de cab1 y la función crear. “*cab1 = lista[*cab1].sgte” permite a la cabeza auxiliar actualizarse cada vez que se crean nuevos nodos y uno por uno, así permite una gestión ordenada de datos y asemejándose a la función “new struct” de la lista dinámica que almacena valor por cada nodo.

- **Insertar dato al inicio y al final de la lista dinámica y lista estática:**

  El objetivo de estas suboperaciones es insertar un nodo al inicio y al final de la lista, posteriormente actualizar la lista.

  Se crea un nuevo nodo en la cual estará el dato solicitado por el usuario.

  Se realizan técnicas lógicas básicas en la lista simple y doble respecto a la base referenciada del último elemento de la lista en nodo siguiente y anterior;

igualmente para las listas simples y dobles circulares respecto a la cabeza(nodo inicial de la lista actualizada).
Se ayuda de nodos auxiliares en los distintos tipos de lista.
Estas suboperaciones sirven de base para las funciones de inserción por posición y dato.

A.1 Insertar dato al inicio en listas dinámicas:

```
void insertarinicio(nodo **lista, int dato){
    nodo *p;
    p=new struct nodo;
    p->dato=dato;
    p->sgte=*lista;
    *lista=p;
    cout<<"Se inserto al inicio de la lista: "<<dato<<endl;
}
```

Está función permite insertar al inicio un dato en “p->dato=dato”, la lista se actualiza en “p->sgte=*lista”, y la cabeza “*lista” apunta al primer nodo recién creado en “*lista=p”.

A.2 Insertar dato al inicio en listas estáticas:

```
void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato){
  int q;
  if(*cab==-1){
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=-1;
    *cab=q;
  }
  else{
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=*cab;
    *cab=q;
```

```
    }
  cout<<dato<<" insertado al inicio de la lista "<<endl;
}
```

A partir de la función declarada y referenciada anteriormente: "nuevonodo(&q,lista,&(*cab1))", permitirá el acceso por cada elemento en la lista presente.

Si "*cab==-1", la lista está vacia y se crea un nuevo nodo, -1 hace referencia a NULL en el trabajo de procesamiento de listas dinámicas. En "lista[q].dato=dato" se inserta el dato, "lista[q].sgte=-1" hace referencia a que el sucesor apunte a NULL. En "*cab=q", la cabeza "*cab" apunta al nodo estático "q" que está situado en el primer elemento de la lista.

Si cab posee otro valor positivo; en "lista[q].dato=dato" se inserta el dato, "lista[q].sgte=*cab" hace referencia a que el sucesor apunte a la cabeza de la lista "*cab". En "*cab=q", la cabeza "*cab" apunta al nodo estático "q" que está situado en el primer elemento de la lista.

B.1 Insertar dato al final de la lista dinámica:

```
void insertarfinal(nodo **lista, int dato){
    nodo *p;
    nodo *q;
    q=new struct nodo;
    if(*lista==NULL){
      q->dato=dato;
      q->sgte=NULL;
      *lista=q;
    }
    else{
      p=*lista;
      q->dato=dato;
      q->sgte=NULL;
      while(p->sgte!=NULL){
        p=p->sgte;
      }
```

```
      p->sgte=q;
    }
    cout<<dato<<" insertado al final de la lista "<<endl;
}
```

Se declara un nodo “q” como el ingreso de un nuevo nodo con “q=new struct nodo”.
Si la lista está vacía, se procede a crear una nuevo dato con “q->dato=dato”, el nodo siguiente apuntando a vacío con “q->sgte=NULL” y la cabeza de la lista apunta al nodo “q” con “*lista=q”.
Si la lista posee elementos, un nodo auxiliar “p” apuntará a la cabeza “lista” con “p=*lista”, se procede a crear un dato nuevo con “q->dato=dato”, el nodo siguiente apuntando a vacío con “q->sgte=NULL”; luego se recorrerá hasta el último nodo de la lista justo antes de NULL con el bucle “while”.
“p->sgte=q”, hace que el anterior último nodo apunte al nuevo último nodo.

B.2 Insertar dato al final de la lista dinámica:

```
void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato){
  int q,p;
  if(*cab==-1){
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=-1;
    *cab=q;
  }
  else{
    nuevonodo(&q,lista,&(*cab1));
    p=*cab;
    lista[q].dato=dato;
    lista[q].sgte=-1;
    while(lista[p].sgte!=-1){
      p=lista[p].sgte;
    }
    lista[p].sgte=q;
```

```
  }
  cout<<dato<<" insertado al final de la lista "<<endl;
}
```

Se declara un nodo “q” como el ingreso de un nuevo nodo con “nuevonodo(&q,lista,&(*cab1))”.

Si la lista está vacía, se procede a crear una nuevo dato con “lista[q].dato=dato”, el nodo siguiente apunta a -1(nulo en listas dinámicas) con “lista[q].sgte=-1” y la cabeza de la lista apunta al nodo “q” con “*cab=q”.

Si la lista posee elementos, un nodo auxiliar “p” apuntará a la cabeza “*cab” con “p=*cab”, se procede a crear un dato nuevo con “lista[q].dato=dato”, el nodo siguiente apunta a -1(nulo en listas dinámicas) con “lista[q].sgte=-1”; luego se recorrerá hasta el último nodo de la lista justo antes de -1(NULL) con el bucle “while”.

“lista[p].sgte=q”, hace que el anterior último nodo apunte al nuevo último nodo.

- **Eliminar dato al inicio y al final de la lista dinámica y lista estática:**

  El objetivo de estas suboperaciones es eliminar un nodo al inicio y al final de la lista y posteriormente actualizar la lista.

  Se eliminará o liberará el nodo pedido por el usuario.

  Se realizan técnicas lógicas básicas en la lista simple y doble respecto a la base referenciada del último elemento de la lista en nodo siguiente y anterior; igualmente para las listas simples y dobles circulares respecto a la cabeza(nodo inicial de la lista actualizada).

  Se ayuda de nodos auxiliares en los distintos tipos de lista.

  Estas suboperaciones sirven de base para las funciones de eliminar por posición y dato.

  A.1 Eliminar dato al inicio de la lista dinámica:

  ```
  void eliminarinicio(nodo **lista){
  ```

```
  int ini, fin;
  if(*lista!=NULL){
    if((*lista)->sgte==NULL){
      *lista=NULL;
    }
    else{
      *lista=(*lista)->sgte;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

Si la lista no está vacía, si posee un nodo aclarándolo con "(*lista)->sgte==NULL", entonces la cabeza de la lista está vacío con "*lista=NULL"; en todo caso, si posee más de un nodo, la cabeza de la lista apuntará al nodo siguiente dejando al primer nodo anterior fuera de referencia con "*lista=(*lista)->sgte". Al recompilar el programa este dato estará libre de la lista principal.

A.2 Eliminar dato al inicio de la lista estática:

```
void eliminarinicio(nodo lista[max], int *cab){
  if(*cab!=-1){
    if(lista[*cab].sgte==-1){
      *cab=-1;
    }
    else{
      *cab=lista[*cab].sgte;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
```

```
    cout<<"Lista vacia"<<endl;
  }
}
```

Si la lista no está vacía, si posee un nodo aclarándolo con “lista[*cab].sgte==-1”, entonces la cabeza de la lista está vacío con “**cab=-1”; en todo caso, si posee más de un nodo, la cabeza de la lista apuntará al nodo siguiente dejando al primer nodo anterior fuera de referencia con “*cab=lista[*cab].sgte”.

B.1 Eliminar dato al final de la lista dinámica:

```
void eliminarfinal(nodo **lista){
  nodo *p;
  nodo *a;
  int ini, fin;
  p=*lista;
  a=*lista;
  if(*lista!=NULL){
    if((*lista)->sgte==NULL){
      *lista=NULL;
    }
    else{
        while(p->sgte!=NULL){
         a=p;
         p=p->sgte;
       }
      a->sgte=NULL;
    }
    cout<<"Se elimino el dato final de la lista"<<endl;
  }
  else{
    cout<<"No existe lista"<<endl;
  }
}
```

Si la lista no está vacía, si posee un nodo aclarándolo con "(*lista)->sgte==NULL", entonces la cabeza de la lista está vacío con "*lista=NULL"; en todo caso, si posee más de un nodo, dentro del bucle "while" recorrerá todos los nodos a partir de "p=p->sgte". Un nodo auxiliar "a" se encargará de llegar al penúltimo nodo de tal manera que enlace a NULL en sustitución del último elemento con "a->sgte=NULL". Al recompilar el programa este dato estará libre de la lista principal.

B.2 Eliminar dato al final de la lista estática:

```
  int p,a;
  p=*cab;
  a=*cab;
  if(*cab!=-1){
    if(lista[*cab].sgte==-1){
      *cab=-1;
    }
    else{
      while(lista[p].sgte!=-1){
        a=p;
        p=lista[p].sgte;
      }
      lista[a].sgte=-1;
    }
    cout<<"Se elimino el dato final de la lista"<<endl;
  }
  else{
    cout<<"No existe lista"<<endl;
  }
}
```

Si la lista no está vacía(-1), si posee un nodo aclarándolo con "lista[*cab].sgte==-1", entonces la cabeza de la lista está vacío con "*cab=-1"; en todo caso, si posee más de un nodo, dentro del bucle "while" recorrerá

todos los nodos a partir de “p=lista[p].sgte”. Un nodo auxiliar “a” se encargará de llegar al penúltimo nodo de tal manera que enlace a -1(NULL) en sustitución del último elemento con “lista[a].sgte=-1”.

- **Insertar dato por posición, antes del dato y después del dato, de la lista dinámica y lista estática:**

  El objetivo de estas suboperaciones es insertar un nodo a la lista y posteriormente ser actualizado, ya sea por posición o al costado de un dato existente de la lista.

  Se toma como base la ubicación de la posición de cada elemento y la cantidad de elementos de la lista, así es más factible realizar las diversas operaciones de inserción de nuevos nodos.

  En estas suboperaciones se resuelven por 3 casos distintos: Una cuando es posición inicial, otra cuando es posición final, y finalmente cuando se usa un procedimiento lógico de inserción de listas para cada una de los distintos tipos de lista(simples, dobles, simples circulares y dobles circulares).

  Se hace énfasis en estas 3 suboperaciones, ya que se pudo reutilizar las suboperaciones de insertar al inicio e insertar al final, para todas las clases de lista y no utilizar los mismos procedimientos otra vez.

A.1 Insertar dato por posición en la lista dinámica:

```
void insertarposicion(nodo **lista, int dato, int pos){
  nodo *p;
  nodo *q;
  nodo *a;
  nodo *b;
  p=new struct nodo;
  int i=1, cant=1, aux;
  q=*lista;
  a=*lista;
  if(pos==1){
```

```
    insertarinicio(&(*lista),dato);
  }
  else {
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    if(pos==cant+1){
      insertarfinal(&(*lista),dato);
    }
    else if(pos<cant+1 && pos>1){
      p->dato=dato;
      if(pos<=cant && pos>1){
        while(i!=pos){
          b=q;
          q=q->sgte;
          i++;
        }
      p->sgte=b->sgte;
      b->sgte=p;
      }
    }
  }
  if(pos<=cant+1 && pos>=1){
    cout<<"Posicion insertada con exito"<<endl;
  }
  else{
    cout<<"Posicion inexistente de la lista"<<endl;
  }
}
```

Si se ingresa en la primera posición con pos=1, entonces trabajará con la función “insertarinicio(&(*lista),dato)”, permitiendo que el dato ingresé con éxito y la lista se actualicé constantemente.

En otro caso, se procede a hallar el número de nodos actuales con el valor "cant" de la lista dentro del bucle "while". Luego, si la posición a insertar es la posición final, entonces trabajará con la función "insertarfinal(&(*lista),dato)" , permitiendo que el dato ingresé con éxito y la lista se actualicé constantemente. Si es una posición entre el primer y el último elemento se crea el nodo "p" para almacenar el dato con "p->dato=dato", y dentro del bucle "while" se procede a almacenar un nodo auxiliar "b" que será el antecesor de otro nodo auxiliar "q", esto servirá para ordenar los nodos adyacentes del nuevo nodo "p" a ingresar y mantenerlos actualizados. Mientras "p->sgte=b->sgte", hace que el nodo "p" apunte al sucesor del nodo adyacente "b"; "b->sgte=p", hace que el sucesor del nodo "b" apunte al nuevo nodo "p", de tal manera que la lista actual quede en los adyacentes consecutivos: nodo b, nuevo nodo p, nodo q.
Si la posición a buscar excede del límite de los nodos de la lista, entonces no se procederá a realizar ninguna operación.

A.2 Insertar dato por posición en la lista estática:

```
void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos){
  int i=1, cant=1 , p, q ,a ,b;
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  if(pos==1){
    insertarinicio(lista, &(*cab), &(*cab1), dato);
  }
  else {
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    if(pos==cant+1){
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else if(pos<cant+1 && pos>1){
```

```
        lista[p].dato=dato;
          while(i!=pos){
            b=q;
            q=lista[q].sgte;
            i++;
          }
        lista[p].sgte=lista[b].sgte;
        lista[b].sgte=p;
      }
    }
    if(pos<=cant+1 && pos>=1){
      cout<<"Posicion insertada con exito"<<endl;
    }
    else{
      cout<<"Posicion inexistente de la lista"<<endl;
    }
}
```

Si se ingresa en la primera posición con pos=1, entonces trabajará con la función “insertarinicio(lista, &(*cab), &(*cab1), dato))”, permitiendo que el dato ingresé con éxito y la lista se actualicé constantemente.

En otro caso, se procede a hallar el número de nodos actuales con el valor “cant” de la lista dentro del bucle “while”. Luego, si la posición a insertar es la posición final, entonces trabajará con la función “insertarfinal(lista, &(*cab), &(*cab1), dato)” , permitiendo que el dato ingresé con éxito y la lista se actualicé constantemente. Si es una posición entre el primer y el último elemento se crea el nodo “p” para almacenar el dato con “lista[p].dato=dato”, y dentro del bucle “while” se procede a almacenar un nodo auxiliar “b” que será el antecesor de otro nodo auxiliar “q”, esto servirá para ordenar los nodos adyacentes del nuevo nodo “p” a ingresar y mantenerlos actualizados. Mientras “lista[p].sgte=lista[b].sgte”, hace que el nodo “p” apunte al sucesor del nodo adyacente “b”; “lista[b].sgte=p”, hace que el sucesor del nodo “b” apunte al nuevo nodo “p”, de tal manera que la

lista actual quede en los adyacentes consecutivos: nodo b, nuevo nodo p, nodo q.

Si la posición a buscar excede del límite de los nodos de la lista, entonces no se procederá a realizar ninguna operación.

B.1 Insertar antes de un dato específico en la lista dinámica:

```
void insertarantesdato(nodo **lista, int dato, int dato2){
  nodo *p;
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  p=new struct nodo;
  p->dato=dato;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos){
      c=c->sgte;
      pos++;
    }
    if(pos==cant){
      if(c->dato!=dato2){
        pos=cant+1;
      }
    }
```

```
    if(pos==1){
      insertarinicio(&(*lista),dato);
    }
    else{
      if(pos<=cant && pos>1){
        while(i!=pos){
          b=q;
          q=q->sgte;
          i++;
        }
      p->sgte=b->sgte;
      b->sgte=p;
    }
    }
    if(pos<=cant && pos>=1){
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }
}
```

El procedimiento es similar a la función anteriormente explicado de “insertar por posición”, adicionalmente se trabajará con el dato ingresado para poder ubicar la posición exacta “pos” de la lista con “while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos)”, en la cual el bucle asegurará que el dato “dato2” a buscar se encuentre, mientras la lista consecutiva no llegue a nulo y la cantidad de nodos de la lista “cant” sea mayor igual a la posición solicitada. Como se trabaja con el ingreso de dato antes de un determinado

segundo dato, entonces no se trabajará con la posición final y el uso de "insertarfinal(&(*lista),dato)".

B.2 Insertar antes de un dato específico en la lista estática:

```
void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){
  int i=1, cant=1, pos=1, p, q, a, b ,c, d;
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[c].dato!=dato2 && cant>=pos){
      c=lista[c].sgte;
      pos++;
    }
    if(pos==1){
      insertarinicio(lista, &(*cab), &(*cab1), dato);
    }
    else{
        if(pos<cant+1 && pos>1){
         lista[p].dato=dato;
         while(i!=pos){
           b=q;
           q=lista[q].sgte;
           i++;
         }
         lista[p].sgte=lista[b].sgte;
         lista[b].sgte=p;
```

```
        }
      }

      if(pos<=cant && pos>=1){
        cout<<dato<<" insertado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }
}
```

El procedimiento es similar a la función anteriormente explicado de "insertar por posición", adicionalmente se trabajará con el dato ingresado para poder ubicar la posición exacta "pos" de la lista con "while(lista[c].dato!=dato2 && cant>=pos)", en la cual el bucle asegurará que el dato "dato2" a buscar se encuentre, mientras la lista consecutiva no llegue a nulo y la cantidad de nodos de la lista "cant" sea mayor igual a la posición solicitada. Como se trabaja con el ingreso de dato antes de un determinado segundo dato, entonces no se trabajará con la posición final y el uso de "insertarfinal(lista, &(*cab), &(*cab1), dato)".

C.1 Insertar después de un dato específico en la lista dinámica:

```
void insertardespuesdato(nodo **lista, int dato, int dato2){
  nodo *p;
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  p=new struct nodo;
  p->dato=dato;
```

```
int i=1, cant=1, pos=1;
q=*lista;
a=*lista;
c=*lista;
if(*lista!=NULL){
  while(a->sgte!=NULL){
    a=a->sgte;
    cant++;
  }
  while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos){
    c=c->sgte;
    pos++;
  }
  if(pos==cant){
    if(c->dato!=dato2){
      pos=cant+1;
    }
  }
  if(pos==cant){
    insertarfinal(&(*lista),dato);
  }
  else{
    if(pos<=cant && pos>=1){
      while(i!=pos){
        b=q;
        q=q->sgte;
        i++;
    }
      p->sgte=q->sgte;
      q->sgte=p;
    }
  }
```

```
    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o
inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}
```

El procedimiento es similar a la función anteriormente explicado de "insertar por posición" e "insertar antes de un dato específico". Como se trabaja con el ingreso de dato después de un determinado segundo dato, entonces no se trabajará con la posición inicial y el uso de "insertarinicio(&(*lista),dato)".

C.2 Insertar después de un dato específico en la lista estática:

```
void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int
dato2){
  int i=1, cant=1, pos=1, p, q, a, b ,c, d;
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[c].dato!=dato2 && cant>=pos){
```

```
      c=lista[c].sgte;
      pos++;
    }
    if(pos==cant){
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else{
      if(pos<=cant && pos>=1){
        d=lista[q].sgte;
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          d=lista[q].sgte;
          i++;
      }
        lista[q].sgte=p;
        lista[p].sgte=d;
      }
    }

    if(pos<=cant && pos>=1){
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }
}
```

El procedimiento es similar a la función anteriormente explicado de “insertar por posición” e “insertar antes de un dato específico”. Como se trabaja con el

ingreso de dato después de un determinado segundo dato, entonces no se trabajará con la posición inicial y el uso de "insertarinicio(lista, &(*cab), &(*cab1), dato)".

- **Eliminar por posición y dato de la lista dinámica y lista estática:**

  El objetivo de estas suboperaciones es eliminar o separar un nodo de la lista y posteriormente ser actualizado, ya sea por posición o un dato existente de la lista.

  Se toma como base la ubicación de la posición de cada elemento y la cantidad de elementos de la lista, así es más factible realizar las diversas operaciones de eliminación de nodos.

  En estas suboperaciones se resuelven por 3 casos distintos: Una cuando es posición inicial, otra cuando es posición final, y finalmente cuando se usa un procedimiento lógico de inserción de listas para cada una de los distintos tipos de lista(simples, dobles, simples circulares y dobles circulares).

  Se hace énfasis en estas 2 suboperaciones, ya que se pudo reutilizar las suboperaciones de eliminar al inicio y eliminar al final para todas las clases de lista y no utilizar los mismos procedimientos otra vez.

  A.1 Eliminar posición en la lista dinámica:

```
void eliminarposicion(nodo **lista, int pos){
  nodo *q;
  nodo *a;
  nodo *b;
  int i=1, cant=1;
  q=*lista;
  a=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    if(pos==1){
```

```
        eliminarinicio(&(*lista));
      }
      else if(pos==cant){
          eliminarfinal(&(*lista));
      }
      else if(pos<cant && pos>1) {
          if(pos<=cant && pos>1){
            while(i!=pos){
              b=q;
              q=q->sgte;
              i++;
          }
          b->sgte=q->sgte;
        }
      }
      if(pos<=cant && pos>=1){
        cout<<"Posicion eliminada con exito"<<endl;
      }
      else{
        cout<<"Posicion inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }
}
```

Se procede a hallar el número de nodos actuales con el valor “cant” de la lista dentro del bucle “while(a->sgte!=NULL)”.

Si se ingresa en la primera posición con pos=1, entonces trabajará con la función “eliminarinicio(&(*lista))”, permitiendo que el dato ingresé con éxito y la lista se actualicé constantemente.

Si se ingresa en la última posición con la cantidad de nodos de la lista “cant”, entonces trabajará con la función “eliminarfinal(&(*lista))”, permitiendo que el dato ingresé con éxito y la lista se actualicé constantemente.

Si es una posición entre el primer y el último elemento, dentro del bucle “while” se procede a almacenar un nodo auxiliar “b” que será el antecesor de otro nodo auxiliar “q”, esto servirá para ordenar los nodos adyacentes y mantenerlos actualizados. “b->sgte=q->sgte”, hace que el sucesor del nodo “b” apunte al sucesor del nodo “q”, de tal manera que la lista actual quede en los adyacentes consecutivos: nodo b, nodo “fuera de la lista”, nodo q.

Si la posición a buscar excede del límite de los nodos de la lista, entonces no se procederá a realizar ninguna operación.

A.2 Eliminar posición en la lista estática:

```
void eliminarposicion(nodo lista[max], int *cab, int pos){
  int i=1, cant=1, p, q, a, b,c;
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    if(pos==1){
      eliminarinicio(lista,&(*cab));
    }
    else if(pos==cant){
        eliminarfinal(lista,&(*cab));
    }
    else if(pos<cant && pos>1) {
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          c=lista[q].sgte;
```

```
            i++;
          }
          lista[b].sgte=c;
        }
        if(pos<=cant && pos>=1){
          cout<<"Posicion eliminada con exito"<<endl;
        }
        else{
          cout<<"Posicion inexistente de la lista"<<endl;
        }
      }
      else{
        cout<<"Lista vacia!"<<endl;
      }
}
```

Se procede a hallar el número de nodos actuales con el valor “cant” de la lista dentro del bucle “while(lista[a].sgte!=-1)”.

Si se ingresa en la primera posición con pos=1, entonces trabajará con la función “eliminarinicio(lista,&(*cab))”, permitiendo que el dato ingresé con éxito y la lista se actualicé constantemente.

Si se ingresa en la última posición con la cantidad de nodos de la lista “cant”, entonces trabajará con la función “eliminarfinal(lista,&(*cab))”, permitiendo que el dato ingresé con éxito y la lista se actualicé constantemente.

Si es una posición entre el primer y el último elemento, dentro del bucle “while” se procede a almacenar un nodo auxiliar “b” que será el antecesor de otro nodo auxiliar “q”, el nodo auxiliar “q” se actualiza a la siguiente posición con “q=lista[q].sgte” esto servirá para ordenar los nodos adyacentes y mantenerlos actualizados. “c=lista[q].sgte”, hace que el nodo “c” apunte al nodo actualizado “q”, de tal manera que la lista actual quede en los adyacentes consecutivos con: nodo “b”, nodo “q(fuera de la lista)”, nodo “c”.

Si la posición a buscar excede del límite de los nodos de la lista, entonces no se procederá a realizar ninguna operación.

B.1 Eliminar dato en la lista dinámica:

```
void eliminardato(nodo **lista, int dato){
  nodo *p;
  nodo *q;
  nodo *a;
  nodo *b;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  p=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    while(p->dato!=dato && p->sgte!=NULL && cant>=pos){
      p=p->sgte;
      pos++;
    }
    if(pos==cant){
      if(p->dato!=dato){
        pos=cant+1;
      }
    }
    if(cant>=pos || pos==1){
      if(pos==1){
        eliminarinicio(&(*lista));
      }
      else {
        if(pos==cant){
          eliminarfinal(&(*lista));
        }
        else{
```

```
            if(pos<cant && pos>1){
              while(i!=pos){
                b=q;
                q=q->sgte;
                i++;
              }
                b->sgte=q->sgte;
              }
            }
          }
        }
        if(pos<=cant && pos>=1){
          cout<<dato<<" eliminado con exito"<<endl;
        }
        else{
          cout<<"Dato inexistente de la lista"<<endl;
        }
      }
      else{
        cout<<"Lista vacia!"<<endl;
      }
}
```

El procedimiento es similar a la función anteriormente explicado de “eliminar por posición”, adicionalmente se trabajará con el dato ingresado para poder ubicar la posición exacta “pos” de la lista con “while(p->dato!=dato && p->sgte!=NULL && cant>=pos)”, en la cual el bucle asegurará que el dato “dato” a buscar se encuentre, mientras la lista consecutiva no llegue a nulo y la cantidad de nodos de la lista “cant” sea mayor igual a la posición solicitada. Se seguirá trabajando con las funciones reutilizables: “eliminarinicio(&(*lista))” y “eliminarfinal(&(*lista))” con la posición inicial y final respectivamente, tomando como base los límites inferior y superior de la lista.

B.2 Eliminar dato en la lista estática:

```
void eliminardato(nodo lista[max], int *cab, int dato){
  int i=1, cant=1, pos=1, p, q, a, b, c;
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[p].dato!=dato && cant>=pos){
      p=lista[p].sgte;
      pos++;
    }
    if(cant>=pos || pos==1){
      if(pos==1){
        eliminarinicio(lista,&(*cab));
      }
      else {
        if(pos==cant){
          eliminarfinal(lista,&(*cab));
        }
        else{
          if(pos<cant && pos>1){
            while(i!=pos){
              b=q;
              q=lista[q].sgte;
              c=lista[q].sgte;
              i++;
            }
            lista[b].sgte=c;
          }
        }
```

```
        }
      }
      if(pos<=cant && pos>=1){
        cout<<dato<<" eliminado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
    }
   else{
      cout<<"Lista vacia!"<<endl;
   }
}
```

El procedimiento es similar a la función anteriormente explicado de “eliminar por posición”, adicionalmente se trabajará con el dato ingresado para poder ubicar la posición exacta “pos” de la lista con “while(lista[p].dato!=dato && cant>=pos)”, en la cual el bucle asegurará que el dato “dato” a buscar se encuentre, mientras la lista consecutiva no llegue a nulo y la cantidad de nodos de la lista “cant” sea mayor igual a la posición solicitada. Se seguirá trabajando con las funciones reutilizables: “eliminarinicio(lista,&(*cab))” y “eliminarfinal(lista,&(*cab))” con la posición inicial y final respectivamente, tomando como base los límites inferior y superior de la lista.

- **Buscar y modificar datos de la lista dinámica y lista estática:**
  El objetivo de estas operaciones es encontrar un dato de la lista.
  Se toma como base la ubicación de la posición de cada elemento y la cantidad de elementos de la lista, así es más factible realizar la búsqueda del dato de la lista.

Estas operaciones son similares en cuanto a procedimiento.

En la operación de buscar un dato, si el dato se encuentra mandará un mensaje de su posición, sino mandará mensaje de inexistencia. En la operación de editar dato es similar a la operación de editar dato, con la diferencia de que la operación de editar dato reemplaza al dato buscado por el dato requerido del usuario y posteriormente actualizará la lista.

A.1 Buscar dato en la lista dinámica:

```
void buscardato(nodo *lista, int dato){
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if(lista!=NULL){
    q=lista;
    while(q->sgte!=NULL){
      q=q->sgte;
      a++;
    }
    p=lista;
    while(p->dato!=dato && p->sgte!=NULL && a>=i){
      p=p->sgte;
      i++;
    }
    if(i==a){
      if(p->dato!=dato){
        i=a+1;
      }
    }
    if(a>=i){
      cout<<dato<<" se encuentra en la posicion: "<<i;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
```

```
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}
```

Si la lista no está vacía, se procede a hallar el número de nodos actuales con la cantidad de nodos “a” y la lista consecutiva con el nodo “q” y “q=q->sgte” de la lista dentro del bucle “while(q->sgte!=NULL)”. Luego, dentro del bucle “while(p->dato!=dato && p->sgte!=NULL && a>=i)”, en la cual el bucle asegurará que el dato “dato” a buscar se encuentre, mientras la lista consecutiva con el nodo “p” y “p=p->sgte” no llegue a nulo y la cantidad de nodos de la lista “a” sea mayor igual a la posición del nodo que se quiere buscar sin excederse de la cantidad permitido. Si el nodo “p” llega al último nodo de la lista, entonces la posición “i” de la lista será mayor que la cantidad de nodos “a”. Esto servirá para las salidas de mensaje si se llegó a encontrar el dato con “a>=i”.

A.2 Buscar dato en la lista estática:

```
void buscardato(nodo lista[max], int cab, int dato){
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=-1){
      cant++;
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){
      pos++;
      p=lista[p].sgte;
    }
  if(cant>=pos && pos>=1){
    cout<<dato<<" se encontro en la posicion "<<pos<<endl;
```

```
  }
  else{
    cout<<"Dato inexistente en la lista"<<endl;
  }
  }
}
```

Si la lista no es -1(vacía), se procede a hallar el número de nodos actuales con la cantidad de nodos “cant” y la lista consecutiva con el nodo “q” y “q=lista[q].sgte” de la lista dentro del bucle “while(lista[q].sgte!=-1)”. Luego, dentro del bucle “while(lista[p].dato!=dato && cant>=pos)”, en la cual el bucle asegurará que el dato “dato” a buscar se encuentre, mientras la lista consecutiva con el nodo “p” y “p=lista[p].sgte” no llegue a nulo y la posición del nodo de la lista “pos” sea mayor igual a la posición del nodo que se quiere buscar sin excederse de la cantidad permitido.

B.1 Modificar dato en la lista dinámica:

```
void editardato(nodo **lista, int dato, int dato2){
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if((*lista)!=NULL){
    q=*lista;
    while(q->sgte!=NULL){ /* Hallo la cantidad de nodos de la lista **/
      q=q->sgte;
      a++;
    }
    p=*lista;
    while(p->dato!=dato && p->sgte!=NULL && a>=i){
      p=p->sgte;
      i++;
    }
    if(i==a){
```

```
        if(p->dato!=dato){
          i=a+1;
        }
      }
      if(p->dato==dato){
        p->dato=dato2;
      }
      if(a>=i){
        cout<<dato<<" se encuentro en la posicion: "<<i<< "y se edito por "<<dato2;
      }
      else{
        cout<<dato<<" no se encuentra en la lista"<<endl;
      }
    }
    else{
      cout<<"La lista esta vacia"<<endl;
    }
}
```

El procedimiento es similar a la función anteriormente explicado de “buscar dato”, adicionalmente se trabajará con el dato ingresado a modificar “dato2” con “p->dato=dato2” y dentro de “if(p->dato==dato)” si el nodo “p” encontró el dato deseado.

B.2 Modificar dato en la lista estática:

```
void editardato(nodo lista[max], int cab, int dato, int dato2){
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=-1){
      cant++;
```

```
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){
      pos++;
      p=lista[p].sgte;
    }
    if(cant>=pos){
      lista[p].dato=dato2;
      cout<<dato<<" se encuentro en la posicion: "<<pos<< " y se edito por
"<<dato2;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
  }
}
```

El procedimiento es similar a la función anteriormente explicado de “buscar dato”, adicionalmente se trabajará con el dato ingresado a modificar “dato2” con “lista[p].dato=dato2” y dentro de “if(cant>=pos)”.

- **Recorrer nodos en la lista dinámica y lista estática:**

  El objetivo de esta operación es recorrer la lista para mostrar los datos en una interfaz deseada. Además, obtener múltiples técnicas de recorrido y su respectivo procedimiento lógico de las listas, que son la base principal de construcción de recorridos posteriores más complejos.

  A.1 Mostrar datos en la lista dinámica:

```
void mostrarlista(nodo *lista){
  nodo *q;
  q=lista;
  if(q!=NULL){
    while(q!=NULL){
      cout<<q->dato<<endl;
      q=q->sgte;
```

```
    }
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

Si la lista no es vacía, dentro del bucle "while(p->sgte!=NULL)", el nodo "p" mostrará toda la lista con "p=p->sgte".

A.2 Mostrar datos en la lista estática:

```
void mostrarlista(nodo lista[max], int cab){
  int q;
  q=cab;
  if(q!=-1){
    while(q!=-1){
      cout<<lista[q].dato<<endl;
      q=lista[q].sgte;
    }
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

Si la lista no es -1(vacía), dentro del bucle "while(q!=-1)", el nodo "p" mostrará toda la lista con "q=lista[q].sgte".

- **Cantidad de elementos de la lista dinámica y lista estática:**

  El objetivo de esta operación es determinar la cantidad de elementos de la lista.

  Se recorrerá todos los nodos de la lista y posteriormente se determinará la cantidad total de elementos de la lista actual.

A.1 Cantidad de datos en la lista dinámica:

```
void cantidad(nodo *lista){
  int cant=1;
  nodo *p;
  p=lista;
  if(lista!=NULL){
    while(p->sgte!=NULL){
      p=p->sgte;
      cant++;
    }
    cout<<"La lista posee: "<<cant<<" elementos"<<endl;
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}
```

Si la lista no es vacía, dentro del bucle “while(p->sgte!=NULL)”, el nodo “p” recorrerá toda la lista con “p=p->sgte” y la cantidad de nodos “cant” se irá actualizando respecto a la lista.

A.2 Cantidad de datos en la lista estática:

```
void cantidad(nodo lista[max], int cab){
  int p, cant=1;
  p=cab;
  if(cab!=-1){
    while(lista[p].sgte!=-1){
      p=lista[p].sgte;
      cant++;
    }
    cout<<"La lista posee: "<<cant<<" elementos"<<endl;
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
```

}

Si la lista no es -1(vacía), dentro del bucle “while(lista[p].sgte!=-1)”, el nodo “p” recorrerá toda la lista con “p=lista[p].sgte” y la cantidad de nodos “cant” se irá actualizando respecto a la lista.

- **Ordenamiento de elementos de la lista dinámica y lista estática:**

  El objetivo de esta operación es ordenar ascendentemente o descendentemente los datos de los nodos de la lista, y posteriormente actualizar la lista.

  Se utilizará diversos métodos de ordenamiento sea búsquedas secuenciales o binarias.

  Para el caso de listas simples y dobles, mi límite será la base referenciada del del último nodo para recorrer toda la lista; en cambio, para el caso de las listas simples y dobles circulares, mi límite será la cantidad de elementos de la lista a partir de la posición y cantidad de elementos de la lista actual.

  A.1 Ordenamiento en la lista dinámica:

```
void ordenardato(nodo *lista){
  int t;
  nodo *p;
  nodo *q;
  p=lista;
  if(lista!=NULL){
    while(p->sgte!=NULL){
      q=p->sgte;
      while(q!=NULL){
        if(p->dato>q->dato){
          t=q->dato;
          q->dato=p->dato;
          p->dato=t;
        }
        q=q->sgte;
```

```
        }
        p=p->sgte;
      }
      cout<<"Lista ordenada ascendentemente"<<endl;
    }
    else{
      cout<<"Lista vacia"<<endl;
    }
}
```

El orden ascendente o descendente de los datos de la lista se puede ordenar tanto numéricamente como en forma de cadenas. Se puede implementar cualquier método de ordenamiento propuesto. Como ejemplo se toma el ordenamiento por burbuja de forma ascendente ayudándose de los nodos auxiliares “p” y “q” y su ventaja de los nodos adyacentes.

A.2 Ordenamiento en la lista estática:

```
void ordenardato(nodo lista[max], int cab){
  int t, p, q;
  p=cab;
  if(cab!=-1){
    while(lista[p].sgte!=-1){
      q=lista[p].sgte;
      while(q!=-1){
        if(lista[p].dato>lista[q].dato){
          t=lista[q].dato;
          lista[q].dato=lista[p].dato;
          lista[p].dato=t;
        }
        q=lista[q].sgte;
      }
      p=lista[p].sgte;
    }
    cout<<"Lista ordenada ascendentemente"<<endl;
```

```
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

- **Guardar y recuperar archivo de la lista dinámica y lista estática:**

  El objetivo de estas suboperaciones es guardar y recuperar el archivo de los datos de las listas con un único nombre de archivo para cada tipo de lista(simples, dobles, simples circulares, dobles circulares).

  Para el caso de listas simples y dobles, en la operación de guardar archivo, mi límite será la base referenciada del último elemento de la lista para recorrer toda la lista y finalmente guardar todos los elementos de la lista en forma de dato.

  Para el caso de listas simples y dobles circulares, en la operación de guardar archivo, mi límite será la cantidad de elementos para recorrer toda la lista y finalmente guardar todos los elementos de la lista en forma de dato.

  En la suboperación de recuperar archivo, se procede a cargar los datos de la lista anteriormente guardada, reutilizando la suboperación de insertar al final, para almacenarlo en forma de nodos de datos de la lista, y también por seguir el orden original y anterior de la última lista guardada.

  A.1 Guardar archivo en la lista dinámica:

```
void guardar(nodo **lista){
  int dato;
  FILE *H;
  nodo *p;
  p=*lista;
  if(p==NULL){
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
  else{
    H=fopen("ListaSim2.dat","w+");
    while(p!=NULL){
```

```
      dato=p->dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=p->sgte;
    }
    fclose(H);
    cout<<"Archivo guardado"<<endl;
  }
}
```

Si existe la lista y posee al menos un nodo entonces se creará un archivo con nombre “ListaSim2.dat” con “H=fopen("ListaSim2.dat","w+")”, se escribirá el archivo con “fwrite(&dato,sizeof(dato),1,H)”. Mientras el archivo exista con el nodo “p” recorriendo toda la lista con “p=p->sgte” y dentro del bucle “while(p!=NULL)”, se procederá a usar la función “fwrite(&dato,sizeof(dato),1,H)” permitiendo guardar en el archivo cada dato “dato” con “dato=p->dato” de su respectivo nodo.

A.2 Guardar archivo en la lista estática:

```
void guardar(nodo lista[max], int cab){
  int p, dato;
  FILE *H;
  p=cab;
  if(p==-1){
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
  else{
    H=fopen("ListaSim.dat","w+");
    while(p!=-1){
      dato=lista[p].dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=lista[p].sgte;
    }
    fclose(H);
    cout<<"Archivo guardado"<<endl;
```

```
  }
}
```

Si existe la lista y posee al menos un nodo entonces se creará un archivo con nombre “ListaSim.dat” con “H=fopen("ListaSim.dat","w+")”, se escribirá el archivo con “fwrite(&dato,sizeof(dato),1,H)”. Mientras el archivo exista con el nodo “p” recorriendo toda la lista con “p=lista[p].sgte” y dentro del bucle “while(p!=-1)”, se procederá a usar la función “fwrite(&dato,sizeof(dato),1,H)” permitiendo guardar en el archivo cada dato “dato” con “dato=lista[p].dato” de su respectivo nodo.

B.1 Recuperar archivo en la lista dinámica:

```
void recuperar(nodo **lista){
  FILE *H;
  int dato;
  H=fopen("ListaSim2.dat","r+");
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
  fread(&dato,sizeof(dato),1,H);
  while(!feof(H)){
    insertarfinal(&(*lista), dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

Si existe la lista y posee al menos un nodo entonces se abrirá un archivo con nombre “ListaSim2.dat” con “H=fopen("ListaSim2.dat","r+")”, se leerá el archivo con “fread(&dato,sizeof(dato),1,H)”. Mientras el archivo exista con el bucle “while(!feof(H))”, se procederá a usar la función

“insertarfinal(&(*lista), dato)” permitiendo cargar cada dato “dato” de su respectivo nodo con “fread(&dato,sizeof(dato),1,H)”.

B.2 Recuperar archivo en la lista estática:

```
void recuperar(nodo lista[max], int *cab, int *cab1){
  int p, dato;
  FILE *H;
  p=*cab;
  H=fopen("ListaSim.dat","r+");
  fread(&dato,sizeof(dato),1,H);
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
  while(!feof(H)){
    insertarfinal(lista,&(*cab),&(*cab1),dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

Si existe la lista y posee al menos un nodo entonces se abrirá un archivo con nombre “ListaSim.dat” con “H=fopen("ListaSim2.dat","r+")”, se leerá el archivo con “fread(&dato,sizeof(dato),1,H)”. Mientras el archivo exista con el bucle “while(!feof(H))”, se procederá a usar la función “insertarfinal(lista,&(*cab),&(*cab1),dato)” permitiendo cargar cada dato “dato” de su respectivo nodo con “fread(&dato,sizeof(dato),1,H)”.

4.3 Realizar aplicaciones de las listas dinámicas y estáticas.

A partir de la aplicación base de las listas simples dinámicas, construir sus extensiones relativas tanto para listas estáticas como listas dinámicas como se muestra en la Tabla 4.2:

Características VS Extensiones	Lista Dinámica Simple	Lista Dinámica Doble	Lista Dinámica Circular Simple	Lista Dinámica Circular Doble
Número de enlaces del nodo de la lista	1 al sucesor.	1 al sucesor y 1 al predecesor.	1 al sucesor.	1 al sucesor y 1 al predecesor.
Recorridos o técnicas eficientes de la lista	Hacia adelante.	Hacia adelante y hacia atrás.	Hacia adelante de manera circular.	Hacia adelante y atrás de manera circular.
Primer Nodo de la Lista	Apunta al primero.	Apunta al primero.	Apunta al primero.	Apunta al primero.
Último Nodo de la Lista	Nulo o vacío.	Nulo o vacío.	Apunta al primero.	Apunta al primero.

Tabla 4.2: Tabla de comparación de las extensiones aplicativas de la lista enlazada.

La descripción de la Tabla 4.2, se especifica por las siguientes características principales tanto para listas dinámicas como las listas estáticas:

- El número de enlaces del nodo de la lista se divide según las categorías de la lista dinámica: lista dinámica simple, doble, circular y circular doble. La lista dinámica simple y la lista dinámica simple circular poseen una lista de nodos en la que cada nodo contiene un solo enlace, anexando al siguiente elemento de la lista. La lista dinámica doble y la lista dinámica circular doble poseen una lista de nodos en la que cada nodo contiene dos enlaces, anexando al anterior y siguiente elemento de la lista.

- Los recorridos o técnicas eficientes de la lista se divide según las categorías de la lista dinámica: lista dinámica simple, doble, circular y circular doble. La lista dinámica simple tiene la técnica de recorrido de nodos de manera unidireccional y secuencial desde el primer nodo hasta el último nodo, denominado recorrido hacia adelante. La lista dinámica simple doble tiene la técnica de recorrido de nodos de manera bidireccional y secuencial desde el primer nodo hasta el último nodo, denominado recorrido hacia adelante y atrás. La lista dinámica simple circular tiene la técnica de recorrido de nodos de manera unidireccional y secuencial desde el primer nodo hasta el mismo primer nodo dando una vuelta completa, denominado recorrido hacia adelante. La lista dinámica simple circular doble tiene la técnica de recorrido de nodos de manera bidireccional y secuencial desde el primer nodo hasta el mismo primer nodo dando una vuelta completa, denominado recorrido hacia adelante y atrás.

- Respecto a la ubicación específica del primer y último nodo de la lista se divide según las categorías de la lista dinámica: lista dinámica simple, doble, circular y circular doble. La lista dinámica simple y la lista dinámica doble tienen al primer nodo referenciado como inicio de la lista y al último nodo anexando a nulo o vacío. La lista dinámica circular simple y la lista dinámica circular doble tienen al primer nodo referenciado como inicio de la lista y al último nodo anexando al primer nodo como siguiente elemento.

- Las bases de las operaciones de las listas simples dinámicas y estáticas se tomarán como referencia para el trabajo de sus extensiones como son las listas dobles, simples circulares y dobles circulares. Las operaciones son semejantes y la única diferencia en las listas simples son como se trabaja el puente en cada nodo adyacente.

- Así como el desarrollo de las listas simples estáticas se basó en las listas simples dinámicas: las listas simples estáticas dobles se basan en las listas simples dinámicas dobles, las listas simples estáticas circulares se basan en las listas simples dinámicas circulares, y las listas dobles estáticas circulares se basan en las listas dobles dinámicas circulares.

4.3.1 Vista panorámica del menú de las listas dobles dinámicas:

El procedimiento de las funciones a realizar se tomará como base la implementación de las operaciones de las listas simples dinámicas con la diferencia que se trabajará con un nodo adicional de la estructura presentada, en vez de un nodo con un dato y un nodo con un enlace siguiente apuntando a otro dato, será un nodo con un enlace anterior apuntando a un dato adyacente y un enlace posterior con otro dato adyacente. (Ver Anexo 4.3.1)

4.3.2 Vista panorámica del menú de las listas dobles estáticas:

El procedimiento de las funciones a realizar se tomará como base la lógica del trabajo de nodos dinámicos de las listas dobles dinámicas, y lo llamaremos nodos estáticos en la implementación de vectores y registros. (Ver Anexo 4.3.2)

4.3.3 Vista panorámica del menú de las listas simples circulares dinámicas:

El procedimiento de las funciones a realizar se tomará como base la implementación de las operaciones de las listas simples dinámicas con la diferencia que el nodo final apunte al nodo inicial. (Ver Anexo 4.3.3)

4.3.4 Vista panorámica del menú de las listas simples circulares estáticas:

El procedimiento de las funciones a realizar se tomará como base la lógica del trabajo de nodos dinámicos de las listas simples circulares dinámicas, y lo llamaremos nodos estáticos en la implementación de vectores y registros. (Ver Anexo 4.3.4)

4.3.5 Vista panorámica del menú de las listas simples dobles circulares dinámicas:

El procedimiento de las funciones a realizar se tomará como base la implementación de las operaciones de las listas simples dinámicas con la diferencia que el nodo final apunte al nodo inicial. (Ver Anexo 4.3.5)

4.3.6 Vista panorámica del menú de las listas simples dobles circulares estáticas:

El procedimiento de las funciones a realizar se tomará como base la lógica del trabajo de nodos dinámicos de las listas simples circulares dinámicas, y lo llamaremos nodos estáticos en la implementación de vectores y registros. (Ver Anexo 4.3.6)

4.4 Realizar pruebas en función tiempo.

- Probar la eficiencia de las listas estáticas en función tiempo en la gestión de miles de cantidades de datos respecto a las listas dinámicas.
- Cada operación individual como insertar, eliminar y sus derivados principalmente; tendrá su tiempo de salida específica por una entrada de cierta cantidad de datos.
- Trabajar con tipos de datos sean principalmente valores numéricos, asociarlos a registros y ver el tiempo de eficiencia de cada uno de ellos.
- Realizar estas pruebas principalmente para el trabajo en función tiempo de las distintas operaciones lógicas listas estáticas y listas dinámicas.
- Se trabajará con muestras de datos de entrada preconfiguradas internamente en el código para la salida directa de tiempo tanto para listas estáticas como listas dinámicas.
- Las funciones principales a probar son de inserción(al inicio, al final, por posición, antes de determinado dato) y de eliminación(al inicio, al final, por posición, por determinado dato).

4.4.1 Inserción

Hay una gran eficiencia de las listas estáticas en el tratamiento de un máximo de 10 000 datos, a partir de ese límite la gestión de datos difiere progresivamente el tiempo en 50 000 y 100 000 datos(con la excepción de insertar al inicio). Se muestra el tiempo de salida por cada cierta cantidad predeterminada de datos y posiciones a insertar, tanto para listas estáticas como listas dinámicas:

A. Insertar varios datos al inicio(Caso General):

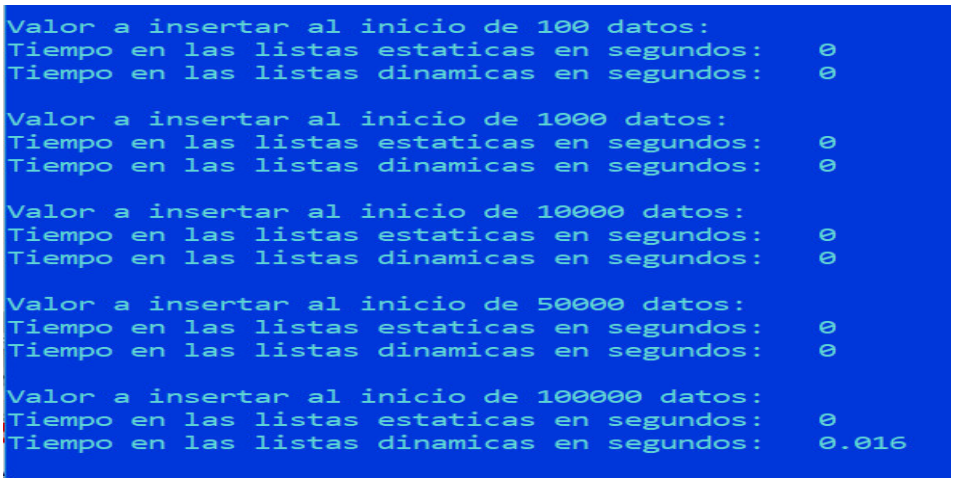

Figura 4.1 Interfaz de inserción de datos al inicio en función tiempo.

B. Insertar varios datos al final(Caso General):

```
Valor a insertar al final de 100 datos:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al final de 1000 datos:
Tiempo en las listas estaticas en segundos:    0.016
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al final de 10000 datos:
Tiempo en las listas estaticas en segundos:    0.169
Tiempo en las listas dinamicas en segundos:    0.184

Valor a insertar al final de 50000 datos:
Tiempo en las listas estaticas en segundos:    4.309
Tiempo en las listas dinamicas en segundos:    4.472

Valor a insertar al final de 100000 datos:
Tiempo en las listas estaticas en segundos:    17.254
Tiempo en las listas dinamicas en segundos:    17.899
```

Figura 4.2 Interfaz de inserción de datos al final en función tiempo.

C. Insertar varios datos en una posición determinada(Caso Automatizado):

```
Valor a insertar al inicio de 100 datos e insercion de 50 posiciones:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al inicio de 1000 datos e insercion de 500 posiciones:
Tiempo en las listas estaticas en segundos:    0.016
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al inicio de 10000 datos e insercion de 5000 posiciones:
Tiempo en las listas estaticas en segundos:    0.252
Tiempo en las listas dinamicas en segundos:    0.263

Valor a insertar al inicio de 50000 datos e insercion de 25000 posiciones:
Tiempo en las listas estaticas en segundos:    6.492
Tiempo en las listas dinamicas en segundos:    6.91

Valor a insertar al inicio de 100000 datos e insercion de 50000 posiciones:
Tiempo en las listas estaticas en segundos:    26.02
Tiempo en las listas dinamicas en segundos:    29.234
```

Figura 4.3 Interfaz de inserción de datos en posiciones determinadas en función tiempo.

D. Insertar varios datos antes de otro dato(Caso Automatizado):

```
Valor a insertar al inicio de 100 datos e insercion de 50 datos:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al inicio de 1000 datos e insercion de 500 datos:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al inicio de 10000 datos e insercion de 5000 datos:
Tiempo en las listas estaticas en segundos:    0.416
Tiempo en las listas dinamicas en segundos:    0.401

Valor a insertar al inicio de 50000 datos e insercion de 25000 datos:
Tiempo en las listas estaticas en segundos:    10.375
Tiempo en las listas dinamicas en segundos:    10.493

Valor a insertar al inicio de 100000 datos e insercion de 50000 datos:
Tiempo en las listas estaticas en segundos:    41.589
Tiempo en las listas dinßmicas en segundos:    46.534
```

Figura 4.4 Interfaz de inserción de datos en datos determinados en función tiempo.

4.4.2 Eliminación

Hay una gran eficiencia de las listas estáticas en el tratamiento de un máximo de 10 000 datos, a partir de ese límite la gestión de datos difiere progresivamente el tiempo en 50 000 y 100 000 datos(con la excepción de eliminar al inicio). Se muestra el tiempo de salida por cada cierta cantidad predeterminada de datos y posiciones a eliminar(extraer los datos originales fuera de la lista), tanto para listas estáticas como listas dinámicas:

A. Eliminar varios datos al inicio(Caso General):

```
Valor a insertar al inicio de 100 datos y eliminacion de 50 datos:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al inicio de 1000 datos y eliminacion de 500 datos:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al inicio de 10000 datos y eliminacion de 5000 datos:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al inicio de 50000 datos y eliminacion de 25000 datos:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0

Valor a insertar al inicio de 100000 datos y eliminacion de 50000 datos:
Tiempo en las listas estaticas en segundos:    0
Tiempo en las listas dinamicas en segundos:    0
```

Figura 4.5 Interfaz de eliminación de datos al inicio en función tiempo.

B. Eliminar varios datos al final(Caso General):

```
Valor a insertar al inicio de 100 datos y eliminacion de 50 datos:
Tiempo en las listas estaticas en segundos:   0
Tiempo en las listas dinamicas en segundos:   0

Valor a insertar al inicio de 1000 datos y eliminacion de 500 datos:
Tiempo en las listas estaticas en segundos:   0
Tiempo en las listas dinamicas en segundos:   0

Valor a insertar al inicio de 10000 datos y eliminacion de 5000 datos:
Tiempo en las listas estaticas en segundos:   0.135
Tiempo en las listas dinamicas en segundos:   0.117

Valor a insertar al inicio de 50000 datos y eliminacion de 25000 datos:
Tiempo en las listas estaticas en segundos:   3.266
Tiempo en las listas dinamicas en segundos:   3.217

Valor a insertar al inicio de 100000 datos y eliminacion de 50000 datos:
Tiempo en las listas estaticas en segundos:   13.067
Tiempo en las listas dinamicas en segundos:   12.65
```

Figura 4.6 Interfaz de eliminación de datos al final en función tiempo.

C. Eliminar varias posiciones determinadas(Caso Automatizado):

```
Valor a insertar al inicio de 100 datos y eliminacion de 50 posiciones:
Tiempo en las listas estaticas en segundos:   0
Tiempo en las listas dinamicas en segundos:   0

Valor a insertar al inicio de 1000 datos y eliminacion de 500 posiciones:
Tiempo en las listas estaticas en segundos:   0
Tiempo en las listas dinamicas en segundos:   0

Valor a insertar al inicio de 10000 datos y eliminacion de 5000 posiciones:
Tiempo en las listas estaticas en segundos:   0.184
Tiempo en las listas dinamicas en segundos:   0.185

Valor a insertar al inicio de 50000 datos y eliminacion de 25000 posiciones:
Tiempo en las listas estaticas en segundos:   4.327
Tiempo en las listas dinamicas en segundos:   4.961

Valor a insertar al inicio de 100000 datos y eliminacion de 50000 posiciones:
Tiempo en las listas estaticas en segundos:   17.581
Tiempo en las listas dinamicas en segundos:   19.736
```

Figura 4.7 Interfaz de eliminación de datos en posiciones determinadas en función tiempo.

D. Eliminar varios datos(Caso Automatizado):

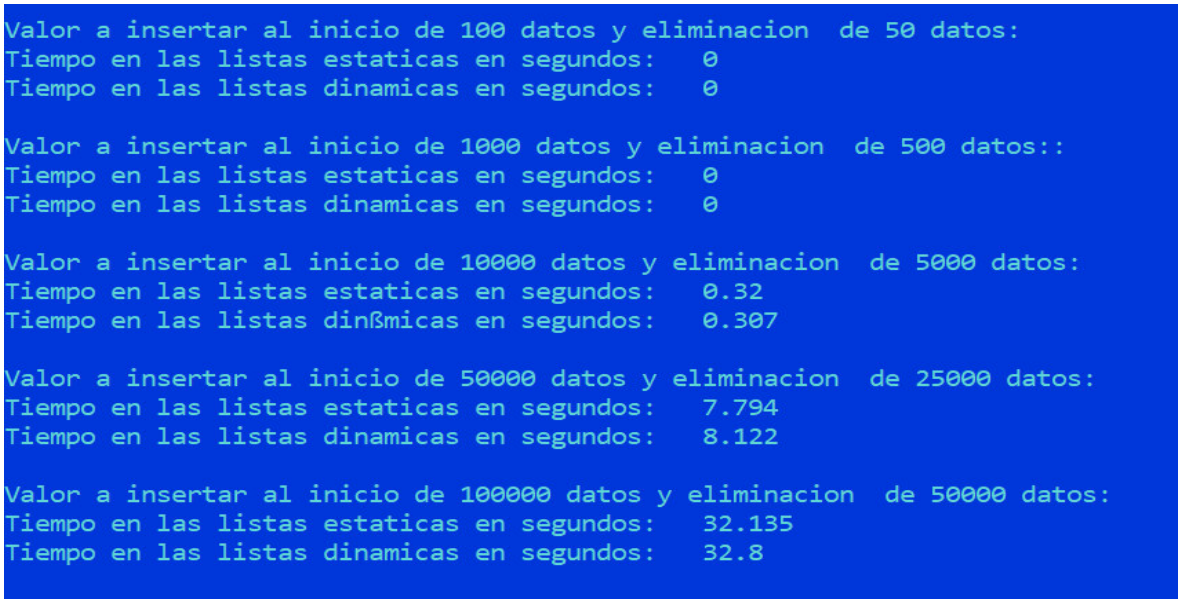

Figura 4.8 Interfaz de eliminación de datos determinados en función tiempo.

4.5 Realizar implementación en código de algoritmos genéticos.

El propósito de la implementación de las listas estáticas es reunir las poblaciones por determinado número de generación, en la cual se procederá a registrar a la población con el individuo que cumple los requisitos en determinado rango.

En un código de algoritmos genéticos[23], se va a extender las funciones de implementación de listas simples estáticas.

El algoritmo base simula la funcionalidad de una muestra al azar de la población con su respectiva limitación.

Sus fases son las siguientes:

A. Generar población.

B. Seleccionar candidatos para el cruce.

C. Cruzar candidatos(con posible probabilidad de mutación).

D. Incluir el mejor de la generación anterior en la nueva generación.

E. Repetir el proceso hasta que cumpla la elección del mejor candidato de la población.

Se tiene el siguiente algoritmo genético que resuelve la minimización de una función y sus respectivas fases, implementado con el modelo de las Listas estáticas. (Ver Anexo 4.5)

Con la funcionalidad de crear, mostrar, insertar al inicio e insertar final de las listas estáticas, se procederá a realizar las siguientes operaciones combinadas y se tendrá los siguientes resultados:

4.5.1 Generación de la población: Se usará la función de crear una nueva lista, de insertar al inicio y mostrar lista.

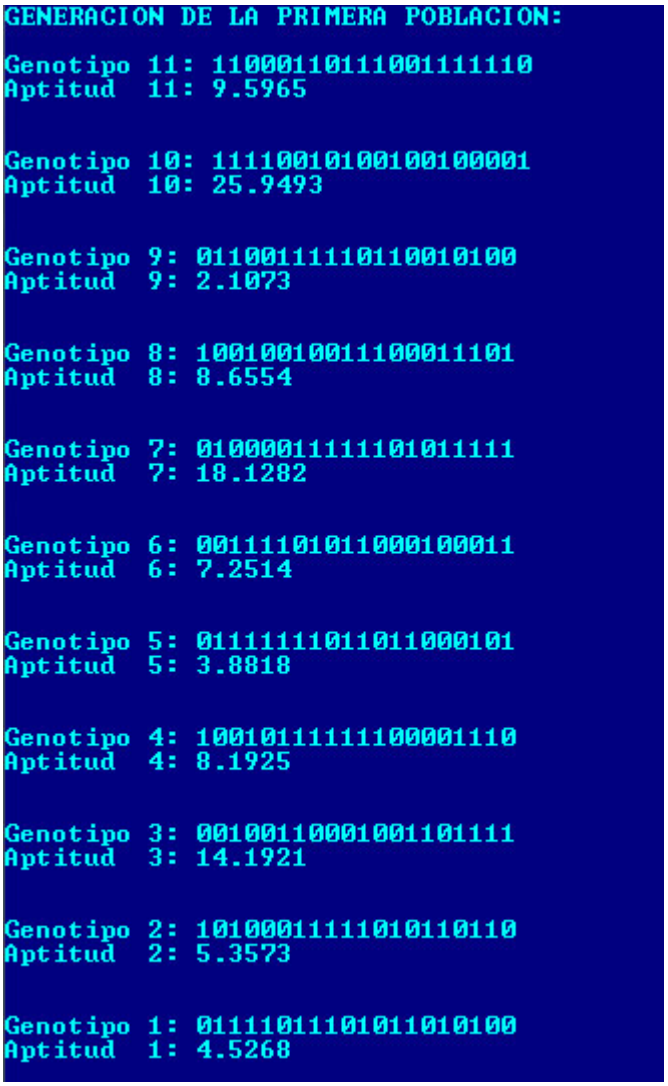

Figura 4.9 Interfaz de inserción de datos al inicio de la primera población.

4.5.2 Selección por Torneos: Se usará la función de crear una nueva lista, de insertar al final y mostrar lista.

Condiciones: La aptitud debe estar entre 0.1 y 1, la generación debe ser diferente de cero, y solo se registrará a la primera población que cumpla dicho requisito.

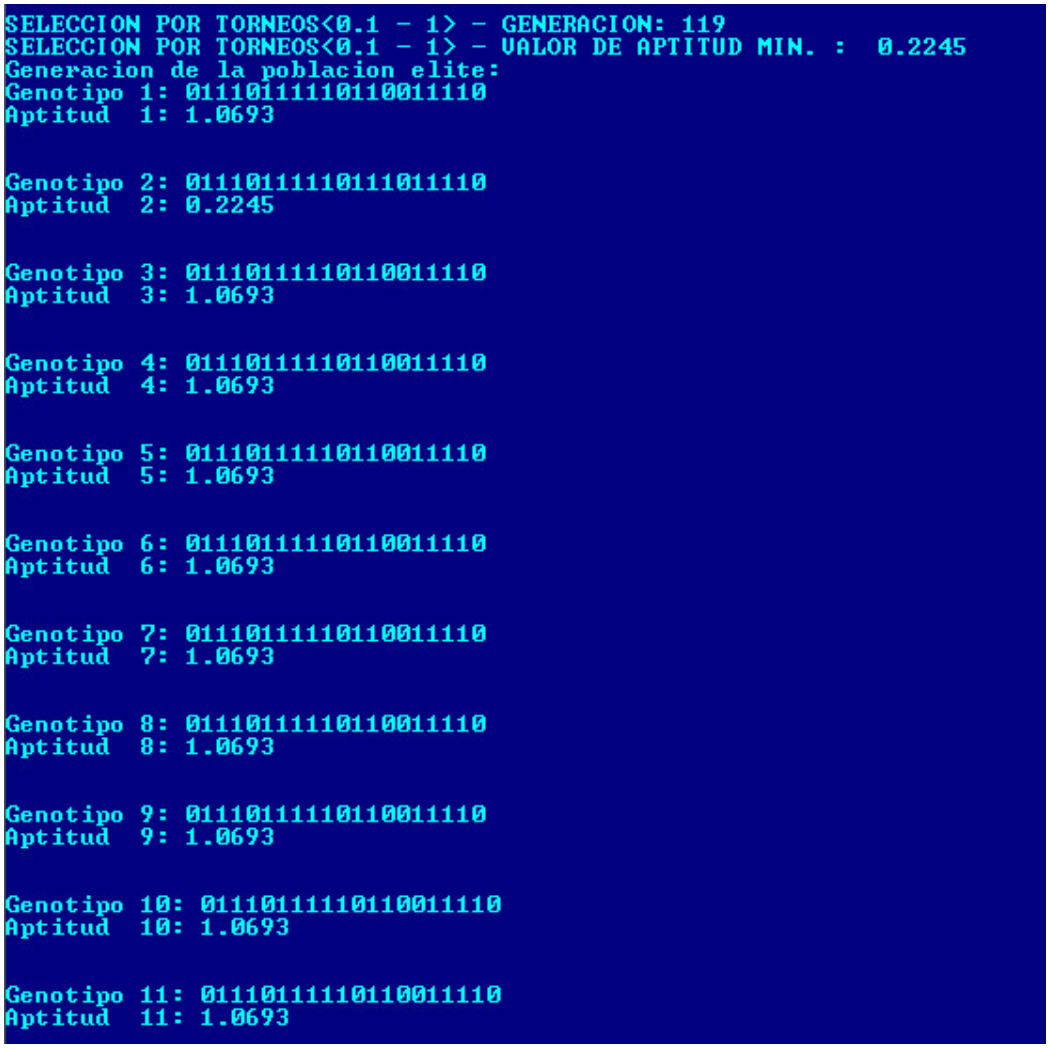

Figura 4.10 Interfaz de inserción de datos al final de Selección de Torneos.

Se mostrará la comprobación del resultado y el punto de valores de la función en la que se encuentra dichos valores:

Figura 4.11 Interfaz de comprobación de puntos de valores de Selección por Torneos.

4.5.3 Cruzamiento de Selección: Se usará la función de crear una nueva lista, de insertar al final y mostrar lista.

Condiciones: La aptitud debe estar entre 1 y 2, la generación debe ser diferente de cero, y solo se registrará a la primera población que cumpla dicho requisito.

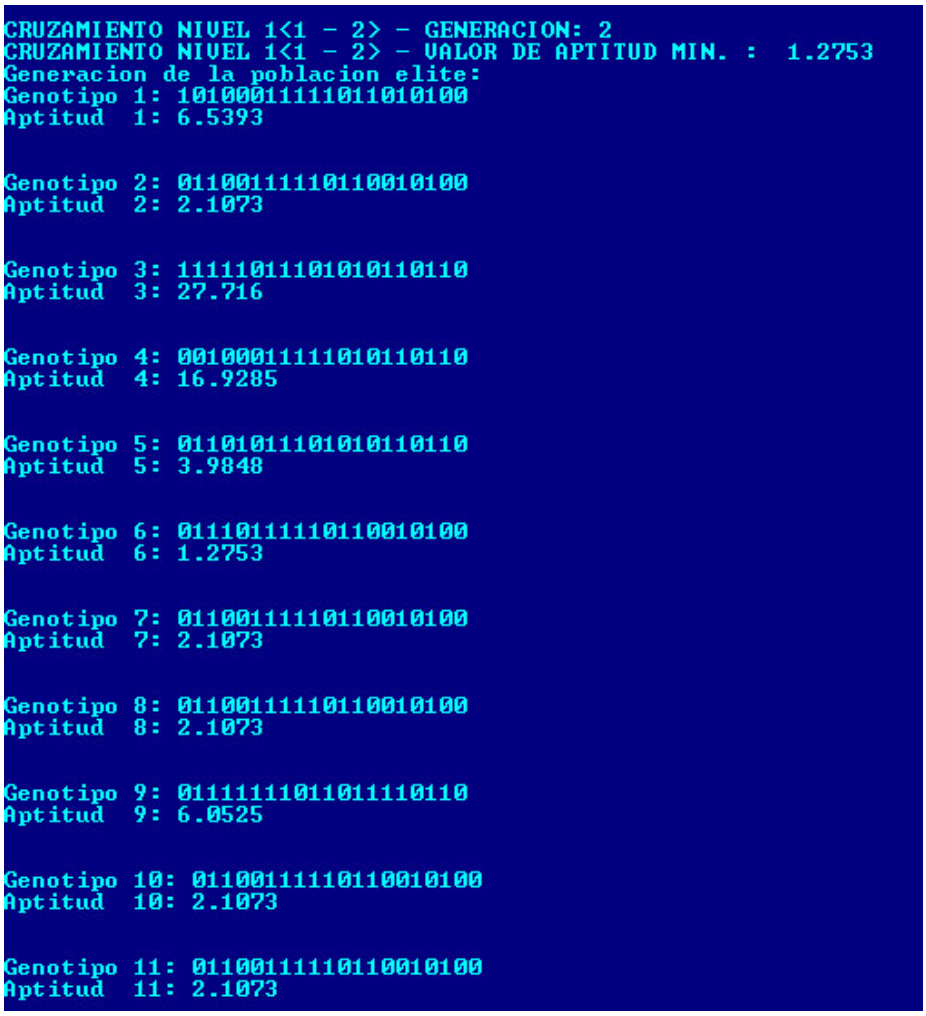

Figura 4.12 Interfaz de inserción de datos al final de Cruzamiento de Selección.

Se mostrará la comprobación del resultado y el punto de valores de la función en la que se encuentra dichos valores:

Figura 4.13 Interfaz de comprobación de puntos de valores de Cruzamiento de Selección.

4.5.4 Cruzamiento de Selección con el mejor de la generación anterior: Se usará la función de crear una nueva lista, de insertar al final y mostrar lista.

Condiciones: La aptitud debe estar entre 2 y 4, la generación debe ser diferente de cero, y solo se registrará a la primera población que cumpla dicho requisito.

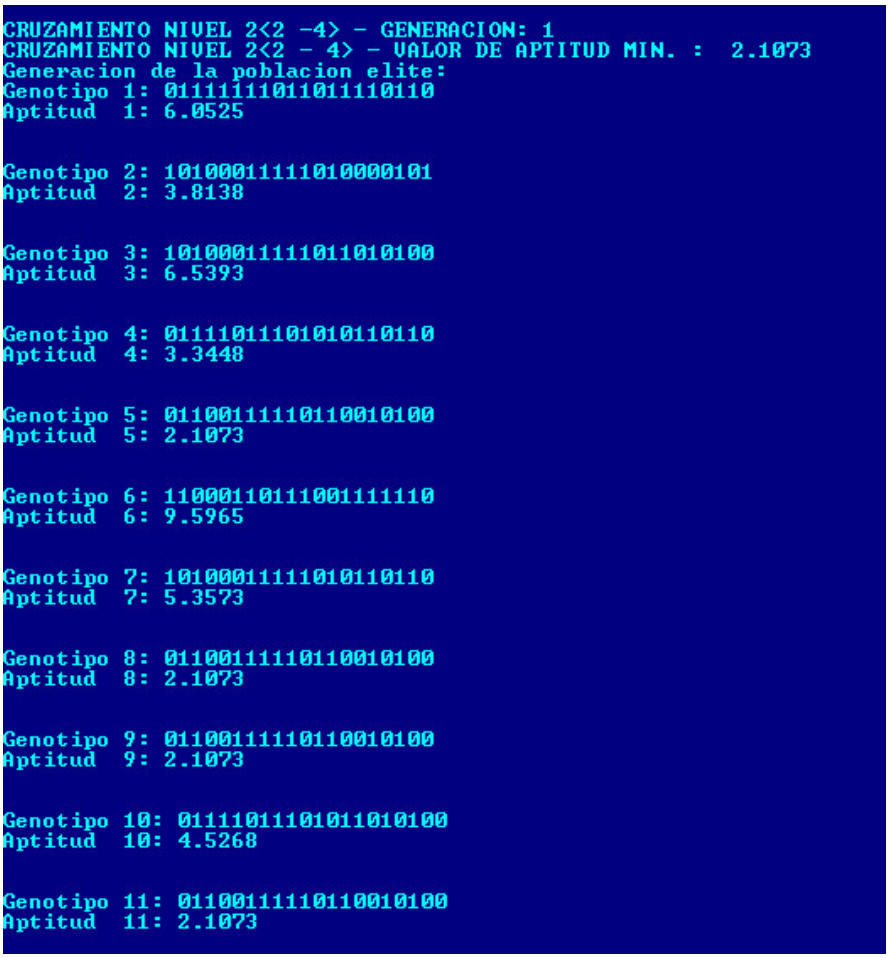

Figura 4.14 Interfaz de inserción de datos al final de Cruzamiento de Selección con el mejor de la generación anterior.

Se mostrará la comprobación del resultado y el punto de valores de la función en la que se encuentra dichos valores:

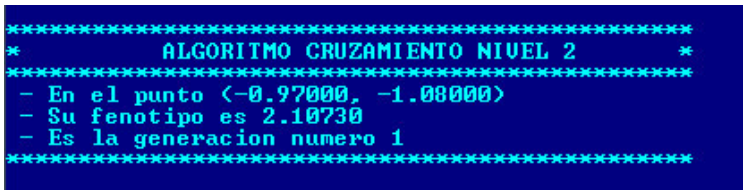

Figura 4.15 Interfaz de comprobación de puntos de valores de Cruzamiento de Selección con el mejor de la generación anterior.

4.5.5 Población ganadora: Se usará la función de crear una nueva lista, de insertar al inicio y mostrar lista.

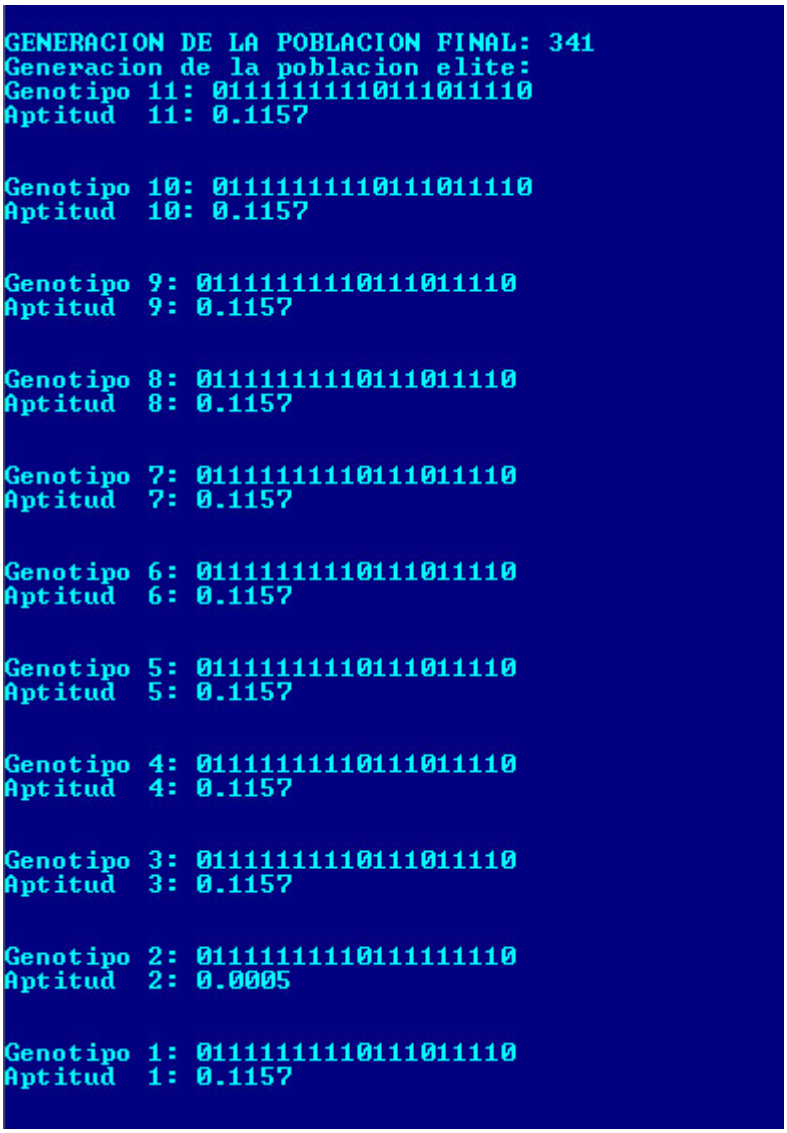

Figura 4.16 Interfaz de inserción de datos al inicio de la población final.

Se mostrará la comprobación del resultado y el punto de valores de la función en la que se encuentra dichos valores:

Figura 4.17 Interfaz de comprobación de puntos de valores de la población final.

CAPÍTULO V. CONCLUSIONES Y RECOMENDACIONES

- La adaptación y homologación de las operaciones de las listas estáticas es en base a las operaciones de listas dinámicas; a fin de que las listas estáticas contengan todas las operaciones posibles(crear, leer, insertar, eliminar, ordenar, etc.) de gestión de los datos.
- El funcionamiento interno del código funciona en base a la estructura de vectores y registros; usando la lógica de la creación, inserción, eliminación de las listas dinámicas.
- La propuesta del modelo de las listas estáticas sirve como un modelo adicional al campo de la estructura de datos, y se considerá una combinación tanto de vectores y registros como las listas dinámicas.
- En las comparaciones y análisis en las operaciones de listas estáticas con listas dinámicas en función tiempo con una predeterminada cantidad de datos, se puede observar que la gestión de datos en las listas estáticas posee un mayor rendimiento que la gestión de las listas dinámicas entre 10 000 y 100 000 datos en la mayoría de casos prueba.
- En el análisis de tiempo de las listas estáticas con las listas dinámicas, los casos automatizados en inserción y eliminación de diferentes posiciones y datos son los casos simulados mayormente usados en los bases de datos modernos.
- En el análisis de tiempo, la ventaja de las listas estáticas es que posee las propiedades de los vectores, la cual les hace poseer una mayor estabilidad que las listas dinámicas, y trabaja eficientemente con una organización de sistemas pequeños de datos.
- El gestionador o controlador de datos universales desarrollado es un menú universal(inserción, eliminación, búsqueda, editado, guardado y recuperado) para poder realizar la correcta y estandarizada gestión de datos tanto para listas dinámicas como listas estáticas con sus respectivos casos: simples, dobles, simples circulares, dobles circulares; adicionalmente se logró especificar detalladamente cómo realizar los procesos lógicos en cada función y operación de la gestión de datos en base a mis reglas generales.
- Implementar listas estáticas(simples, dobles, simples circulares, dobles circulares) es una organizada, parametrizada y alternativa versión de

implementar listas dinámicas ya que sirve como una introducción para el entendimiento de las listas dinámicas, las operaciones de listas estáticas gira en torno de la creación del nodo para un mayor entendimiento de las listas dinámicas.

- Las propiedades de las listas simples, como son sus múltiples casos: simples, dobles, simples circulares, dobles circulares; toma como base principal la extensión del desarrollo multifuncional de manipulación de datos, tanto para las listas estáticas como las listas dinámicas.
- El funcionamiento interno de las pilas se puede considerar como una combinación recíproca de sus respectivas operaciones principales: "apilar" y "desapilar" dato; equivalente a la implementación tanto listas estáticas como listas dinámicas: funciones de "insertar al inicio" y "eliminar al inicio" respectivamente.
- El funcionamiento interno de las colas se puede considerar como una combinación recíproca de sus respectivas operaciones principales: "encolar" y "decolar" dato; equivalente a la implementación tanto listas estáticas como listas dinámicas: funciones de "insertar al final" y "eliminar al inicio" respectivamente.
- Las fases de algoritmos genéticos pueden ser distribuidas y almacenadas por las listas estáticas, ya sea por el primero que cumple una determinada aptitud, está en un determinado rango de valores, o poseer varias listas con varias poblaciones específicas por restricciones propuestas.
- Los algoritmos genéticos y sus fases se pueden proponer de manera libre por el autor, teniendo en cuenta la problemática del ejercicio o modelo propuesto, y las restricciones o delimitaciones respectivas por las múltiples variables declaradas.
- Como recomendación se podría ampliar y expandir el funcionamiento de las listas estáticas a árboles y grafos estáticos.
- Otra recomendación es proponer aplicaciones e implementación externa en las diferentes bases de datos de datos de la actualidad; y destacar óptimos resultados en respuesta de tiempo, respecto a los múltiples casos de los algoritmos genéticos.

CAPÍTULO VI. ANEXOS

ANEXO 4.1.1

4.1.1 Vista panorámica del menú de las listas simples dinámicas:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

using namespace std;

struct nodo{ /** Declaro una estructura en forma de lista simple dinámica **/
  int dato;
  nodo *sgte;
  };

void crearlista(nodo **lista); /** Funciones cabeceras **/
void mostrarlista(nodo *lista);
void insertarinicio(nodo **lista, int dato);
void insertarfinal(nodo **lista, int dato);
void insertarposicion(nodo **lista, int dato, int pos);
void insertarantesdato(nodo **lista, int dato, int dato2);
void insertardespuesdato(nodo **lista, int dato, int dato2);
void eliminarinicio(nodo **lista);
void eliminarfinal(nodo **lista);
void eliminarposicion(nodo **lista, int pos);
void eliminardato(nodo **lista, int dato);
void buscardato(nodo *lista, int dato);
void editardato(nodo **lista, int dato, int dato2);
void ordenardato(nodo *lista);
void cantidad(nodo *lista);
void guardar(nodo **lista);
void recuperar(nodo **lista);
```

```
int main(){
   nodo *lista;
   crearlista(&lista);
   int opc, dato, dato2, pos;
   do{
      system("color 1B");
      //system("cls");
      cout<<"\n\n\n";
      cout<<("*****MENU*****")<<endl; /** Por aquí se visualiza el MENU **/
      cout<<"1.  Mostrar lista"<<endl;
      cout<<"2.  Insertar al inicio"<<endl;
      cout<<"3.  Insertar al final"<<endl;
      cout<<"4.  Insertar en una posicion determinada"<<endl;
      cout<<"5.  Insertar antes de un dato"<<endl;
      cout<<"6.  Insertar despues de un dato"<<endl;
      cout<<"7.  Eliminar al inicio"<<endl;
      cout<<"8.  Eliminar al final"<<endl;
      cout<<"9.  Eliminar en una posicion determinada"<<endl;
      cout<<"10. Eliminar un dato"<<endl;
      cout<<"11. Buscar dato"<<endl;
      cout<<"12. Editar dato"<<endl;
      cout<<"13. Ordenar lista"<<endl;
      cout<<"14. Cantidad de datos"<<endl;
      cout<<"15. Guardar lista"<<endl;
      cout<<"16. Recuperar lista"<<endl;
      cout<<"17. Salir"<<endl;
      cout<<"Ingrese opcion: ";
      cin>>opc;
      cout<<"\n\n\n";
      switch(opc){
        case 1:  cout<<"LISTADO: "<<endl;
                 mostrarlista(lista);
                 break;
```

```
case 2:  cout<<"Valor a insertar al inicio: ";
        cin>>dato;
        insertarinicio(&lista, dato);
        break;
case 3:  cout<<"Valor a insertar al final: ";
        cin>>dato;
        insertarfinal(&lista, dato);
        break;
case 4:  cout<<"Valor a insertar: ";
        cin>>dato;
        cout<<"En que posicion?: ";
        cin>>pos;
        insertarposicion(&lista, dato, pos);
        break;
case 5:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Antes del dato: ";
        cin>>dato2;
        insertarantesdato(&lista, dato, dato2);
        break;
case 6:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Despues del dato: ";
        cin>>dato2;
        insertardespuesdato(&lista, dato, dato2);
        break;
case 7:  eliminarinicio(&lista);
        break;
case 8:  eliminarfinal(&lista);
        break;
case 9:  cout<<"Posicion a eliminar: ";
        cin>>pos;
        eliminarposicion(&lista, pos);
        break;
```

```
        case 10: cout<<"Dato a eliminar: ";
                 cin>>dato;
                 eliminardato(&lista, dato);
                 break;
        case 11: cout<<"Dato a buscar: ";
                 cin>>dato;
                 buscardato(lista, dato);
                 break;
        case 12: cout<<"Dato a buscar: ";
                 cin>>dato;
                 cout<<"Editar por: ";
                 cin>>dato2;
                 editardato(&lista, dato, dato2);
                 break;
        case 13: ordenardato(lista);
                 break;
        case 14: cantidad(lista);
                 break;
        case 15: guardar(&lista);
                 break;
        case 16: recuperar(&lista);
                 break;
      }
   }
   while(opc!=17);
   system("pause");
   return(0);
}


void crearlista(nodo **lista){
   *lista=NULL;
}
```

```
void mostrarlista(nodo *lista){ /** Muestra cada nodo visualizado a partir de las estructuras creadas hasta el ultimo momento **/
  nodo *q;
  q=lista;
  if(q!=NULL){
    while(q!=NULL){ /** Recorre todos los nodos **/
      cout<<q->dato<<endl;
      q=q->sgte;
    }
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void insertarinicio(nodo **lista, int dato){ /** Se inserta dato al inicio de la lista **/
    nodo *p; /** Nodos auxiliares **/
    p=new struct nodo;
    p->dato=dato;
    p->sgte=*lista;
    *lista=p; /** Al final la cabeza lista apuntará al nodo creado con el dato ingresado **/
    cout<<"Se inserto al inicio de la lista: "<<dato<<endl;
}


void insertarfinal(nodo **lista, int dato){
    nodo *p;
    nodo *q;
    q=new struct nodo;
    if(*lista==NULL){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a NULL(-1) **/
```

```
        q->dato=dato;
        q->sgte=NULL;
        *lista=q;
      }
      else{ /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el ultimo nodo-sgte apunten a NULL(-1) **/
        p=*lista;
        q->dato=dato;
        q->sgte=NULL;
        while(p->sgte!=NULL){
          p=p->sgte;
        }
        p->sgte=q;
      }
      cout<<dato<<" insertado al final de la lista "<<endl;
}


void insertarposicion(nodo **lista, int dato, int pos){
   nodo *p; /**Nodos auxiliares **/
   nodo *q;
   nodo *a;
   nodo *b;
   p=new struct nodo;
   int i=1, cant=1, aux;
   q=*lista;
   a=*lista;
   if(pos==1){ /** Si la lista está vacia o se inserta a la 1° posición se llama a la función insertarinicio **/
      insertarinicio(&(*lista),dato);
   }
   else {
      while(a->sgte!=NULL){ /** Si la lista posee más de un elemento se halla la cantidad de nodos en la lista **/
```

```
      a=a->sgte;
      cant++;
    }
    if(pos==cant+1){ /** Si se quiere insertar en la posición sgte. de toda la lista se llama a la función insertarfinal **/
      insertarfinal(&(*lista),dato);
    }
    else if(pos<cant+1 && pos>1){
      p->dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
      if(pos<=cant && pos>1){
        while(i!=pos){
          b=q;
          q=q->sgte;
          i++;
        }
      p->sgte=b->sgte;
      b->sgte=p;
      }
    }
  }
  if(pos<=cant+1 && pos>=1){ /** Si la posición está entre los rangos normales de la lista se procede a enviar un mensaje que se insertó **/
    cout<<"Posicion insertada con exito"<<endl;
  }
  else{ /** En todo caso no realiza ninguna operación por estar fuera de los rangos de la lista **/
    cout<<"Posicion inexistente de la lista"<<endl;
  }
}


void insertarantesdato(nodo **lista, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
```

```
  nodo *p;                    /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  p=new struct nodo;
  p->dato=dato;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=c->sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){
      if(c->dato!=dato2){
        pos=cant+1;
      }
    }
    if(pos==1){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarinicio(&(*lista),dato);
    }
    else{
      if(pos<=cant && pos>1){
        while(i!=pos){
```

```
            b=q;
            q=q->sgte;
            i++;
           }
        p->sgte=b->sgte;
        b->sgte=p;
      }
      }
      if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
        cout<<dato<<" insertado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
   }
   else{
      cout<<"Lista vacia!"<<endl;
   }
}



void insertardespuesdato(nodo **lista, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
   nodo *p;                          /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
   nodo *q;
   nodo *a;
   nodo *b;
   nodo *c;
   p=new struct nodo;
   p->dato=dato;
   int i=1, cant=1, pos=1;
```

```
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=c->sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){
      if(c->dato!=dato2){
        pos=cant+1;
      }
    }
    if(pos==cant){   /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarfinal(&(*lista),dato);
    }
    else{
      if(pos<=cant && pos>=1){
        while(i!=pos){
          b=q;
          q=q->sgte;
          i++;
      }
        p->sgte=q->sgte;
        q->sgte=p;
      }
    }
```

```
    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}




void eliminarinicio(nodo **lista){
  if(*lista!=NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if((*lista)->sgte==NULL){ /** Si la lista es una sola, entoces apuntará a NULL **/
      *lista=NULL;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo **/
      *lista=(*lista)->sgte;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

```
void eliminarfinal(nodo **lista){
  nodo *p;
  nodo *a;
  p=*lista;
  a=*lista;
  if(*lista!=NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if((*lista)->sgte==NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
      *lista=NULL;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo **/
      while(p->sgte!=NULL){
        a=p;
        p=p->sgte;
      }
      a->sgte=NULL;
    }
    cout<<"Se elimino el dato final de la lista"<<endl;
  }
  else{
    cout<<"No existe lista"<<endl;
  }
}


void eliminarposicion(nodo **lista, int pos){
  nodo *q;
  nodo *a;
  nodo *b;
  int i=1, cant=1;
  q=*lista;
```

```
  a=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){ /** Se halla la cantidad de nodos/datos de la lista **/
      a=a->sgte;
      cant++;
    }
    if(pos==1){ /** Si la posicion a eliminar es inicial llamar a eliminarinicio **/
      eliminarinicio(&(*lista));
    }
    else if(pos==cant){ /** Si la posicion es la posicion final de la lista llamar a eliminarfinal **/
        eliminarfinal(&(*lista));
    }
    else if(pos<cant && pos>1) { /** Hallo un caso general para cualquier posicion que se encuentre entre la primera posición y ´ltima posición **/
        if(pos<=cant && pos>1){
          while(i!=pos){
            b=q;
            q=q->sgte;
            i++;
        }
        b->sgte=q->sgte;
      }
    }
    if(pos<=cant && pos>=1){ /** Mandar mensaje si se cumplió que la posición está dentro de los rangos de la lista **/
      cout<<"Posicion eliminada con exito"<<endl;
    }
    else{ /** En todo caso no se pudó realizar ninguna operación y mandar mensaje **/
      cout<<"Posicion inexistente de la lista"<<endl;
    }
  }
  else{
```

```
    cout<<"Lista vacia!"<<endl;
  }
}


void eliminardato(nodo **lista, int dato){ /** Similar a la funcion eliminarposicion **/
  nodo *p;                          /** Solo ubicaré la posición del dato y el resto lo haré como la función eliminarposicion **/
  nodo *q;
  nodo *a;
  nodo *b;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  p=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    while(p->dato!=dato && p->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      p=p->sgte;
      pos++;
    }
    if(pos==cant){
      if(p->dato!=dato){
        pos=cant+1;
      }
    }
    if(cant>=pos || pos==1){
      if(pos==1){           /** Este procedimiento a seguir es igual a la de la funcion eliminarposicion **/
```

```
        eliminarinicio(&(*lista));
      }
      else {
        if(pos==cant){
          eliminarfinal(&(*lista));
        }
        else{
          if(pos<cant && pos>1){
            while(i!=pos){
              b=q;
              q=q->sgte;
              i++;
          }
            b->sgte=q->sgte;
          }
        }
      }
    }
    if(pos<=cant && pos>=1){ /** Mensajes de aviso si el dato existe o no **/
      cout<<dato<<" eliminado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}



void buscardato(nodo *lista, int dato){
```

```
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if(lista!=NULL){
    q=lista;
    while(q->sgte!=NULL){ /** Hallo la cantidad de nodos de la lista **/
      q=q->sgte;
      a++;
    }
    p=lista;
    while(p->dato!=dato && p->sgte!=NULL && a>=i){ /** Realizo operaciones para buscar el dato **/
      p=p->sgte;
      i++;
    }
    if(i==a){
      if(p->dato!=dato){ /** Si el nodo p llega al último dato **/
        i=a+1;
      }
    }
    if(a>=i){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      cout<<dato<<" se encuentra en la posicion: "<<i;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}
```

```
void editardato(nodo **lista, int dato, int dato2){
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if((*lista)!=NULL){
    q=*lista;
    while(q->sgte!=NULL){ /** Hallo la cantidad de nodos de la lista **/
      q=q->sgte;
      a++;
    }
    p=*lista;
    while(p->dato!=dato && p->sgte!=NULL && a>=i){ /** Realizo operaciones para buscar el dato **/
      p=p->sgte;
      i++;
    }
    if(i==a){
      if(p->dato!=dato){ /** Si el nodo p llega al último dato **/
        i=a+1;
      }
    }
    if(p->dato==dato){ /** Si el p->dato es igual al dato encontrado entonces se modifica **/
      p->dato=dato2;
    }
    if(a>=i){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      cout<<dato<<" se encuentro en la posicion: "<<i<< "y se edito por "<<dato2;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
  }
  else{
```

```
    cout<<"La lista esta vacia"<<endl;
  }
}


void cantidad(nodo *lista){
  int cant=1;
  nodo *p;
  p=lista;
  if(lista!=NULL){
    while(p->sgte!=NULL){ /** Cuenta los elementos yendo a través de cada dato
**/
      p=p->sgte;
      cant++;
    }
    cout<<"La lista posee: "<<cant<<" elementos"<<endl;
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }

}


void ordenardato(nodo *lista){ /** Ordenamiento por el método de burbuja asociada
a listas **/
  int t;
  nodo *p;
  nodo *q;
  p=lista;
  if(lista!=NULL){
    while(p->sgte!=NULL){
      q=p->sgte;
      while(q!=NULL){
```

```
            if(p->dato>q->dato){
              t=q->dato;
              q->dato=p->dato;
              p->dato=t;
            }
            q=q->sgte;
          }
          p=p->sgte;
        }
        cout<<"Lista ordenada ascendentemente"<<endl;
      }
      else{
        cout<<"Lista vacia"<<endl;
      }
}


void guardar(nodo **lista){ /** Guardar el archivo de la lista simple en forma de dato **/
  int dato;
  FILE *H;
  nodo *p;
  p=*lista;
  if(p==NULL){
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
  else{
    H=fopen("ListaSim2.dat","w+");
    while(p!=NULL){
      dato=p->dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=p->sgte;
    }
    fclose(H);
```

```
    cout<<"Archivo guardado"<<endl;
  }
}

void recuperar(nodo **lista){ /** Se recuperan los datos y se insertar al final en una lista simple **/
  FILE *H;
  int dato;
  H=fopen("ListaSim2.dat","r+");
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
  fread(&dato,sizeof(dato),1,H);
  while(!feof(H)){
    insertarfinal(&(*lista), dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

ANEXO 4.1.2

4.1.2 Vista panorámica del menú de las listas simples estáticas:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define max 100 /** Define la cantidad máxima de listas estáticas con la que voy a trabajar **/

using namespace std;

struct nodo{ /** Declaro una estructura en forma de lista simple estática **/
  int dato;
  int sgte;
};

void crear(nodo lista[max], int *cab, int *cab1); /** Funciones cabeceras **/
void nuevonodo(int *p, nodo lista[max], int *cab1);
void mostrarlista(nodo lista[max], int cab);
void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato);
void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato);
void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos);
void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2);
void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2);
void eliminarinicio(nodo lista[max], int *cab);
void eliminarfinal(nodo lista[max], int *cab);
void eliminarposicion(nodo lista[max], int *cab, int pos);
void eliminardato(nodo lista[max], int *cab, int dato);
void buscardato(nodo lista[max], int cab, int dato);
void editardato(nodo lista[max], int cab, int dato, int dato2);
void ordenardato(nodo lista[max], int cab);
void cantidad(nodo lista[max], int cab);
void guardar(nodo lista[max], int cab);
void recuperar(nodo lista[max], int *cab, int *cab1);
```

```
int main(){
  nodo lista[max]; /** Declaro una variable de la lista estática simple **/
  int cab, cab1, opc, dato, dato2, pos; /** La cabecera de la lista será una variable cab
**/
  crear(lista,&cab,&cab1); /** cab1 ayudará a la creación de nuevos nodos **/
  do{
    system("color 1B");
    cout<<"\n\n\n";
    cout<<("*****MENU*****")<<endl; /** Por aquí se visualiza el MENU **/
    cout<<"1.  Mostrar lista"<<endl;
    cout<<"2.  Insertar al inicio"<<endl;
    cout<<"3.  Insertar al final"<<endl;
    cout<<"4.  Insertar en una posicion determinada"<<endl;
    cout<<"5.  Insertar antes de un dato"<<endl;
    cout<<"6.  Insertar despues de un dato"<<endl;
    cout<<"7.  Eliminar al inicio"<<endl;
    cout<<"8.  Eliminar al final"<<endl;
    cout<<"9.  Eliminar en una posicion determinada"<<endl;
    cout<<"10. Eliminar un dato"<<endl;
    cout<<"11. Buscar dato"<<endl;
    cout<<"12. Editar dato"<<endl;
    cout<<"13. Ordenar lista"<<endl;
    cout<<"14. Numero de elementos de la lista"<<endl;
    cout<<"15. Guardar lista"<<endl;
    cout<<"16. Recuperar lista"<<endl;
    cout<<"17. Salir"<<endl;
    cout<<"Ingrese opcion: ";
    cin>>opc;
    cout<<"\n\n\n";
    switch(opc){
      case 1:  cout<<"LISTADO: "<<endl;
               mostrarlista(lista,cab);
```

```
        break;
case 2:  cout<<"Valor a insertar al inicio: ";
        cin>>dato;
        insertarinicio(lista,&cab,&cab1,dato);
        break;
case 3:  cout<<"Valor a insertar al final: ";
        cin>>dato;
        insertarfinal(lista,&cab,&cab1,dato);
        break;
case 4:  cout<<"Valor a insertar: ";
        cin>>dato;
        cout<<"En que posicion?: ";
        cin>>pos;
        insertarposicion(lista,&cab,&cab1,dato,pos);
        break;
case 5:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Antes del dato: ";
        cin>>dato2;
        insertarantesdato(lista,&cab,&cab1,dato,dato2);
        break;
case 6:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Despues del dato: ";
        cin>>dato2;
        insertardespuesdato(lista,&cab,&cab1,dato,dato2);
        break;
case 7:  eliminarinicio(lista,&cab);
        break;
case 8:  eliminarfinal(lista,&cab);
        break;
case 9:  cout<<"Posicion a eliminar: ";
        cin>>pos;
        eliminarposicion(lista,&cab,pos);
```

```
            break;
        case 10:  cout<<"Dato a eliminar: ";
            cin>>dato;
            eliminardato(lista,&cab,dato);
            break;
        case 11: cout<<"Dato a buscar: ";
            cin>>dato;
            buscardato(lista,cab,dato);
            break;
        case 12: cout<<"Dato a buscar: ";
            cin>>dato;
            cout<<"Editar por: ";
            cin>>dato2;
            editardato(lista, cab, dato, dato2);
            break;
        case 13: ordenardato(lista,cab);
            break;
        case 14: cantidad(lista, cab);
            break;
        case 15: guardar(lista,cab);
            break;
        case 16: recuperar(lista,&cab,&cab1);
            break;
      }
   }
   while(opc!=17);
   system("pause");
   return(0);
}
```

void crear(nodo lista[max], int *cab, int *cab1){ /** Crea todas las estructuras posibles con el límite máximo declarado simulando la creación de espacio de nodos **/

```
  int i;
  *cab =-1;
  *cab1 =0;
  lista[0].sgte =1;
  for(i=1;i<max-1;i++)
  {
    lista[i].dato=0;
    lista[i].sgte = i+1;
  }
  lista[max-1].sgte =-1;
}

void nuevonodo(int *p, nodo lista[max], int *cab1) /** Crea un nuevo nodo a partir de la posicion de cab1 y la función crear **/
{
  *p = *cab1;
  if(*cab1 == -1)
  {
    printf(" No hay suficiente memoria\n\n");
    exit(1);
  }
  *cab1 = lista[*cab1].sgte;
}


void mostrarlista(nodo lista[max], int cab){ /** Muestra cada nodo visualizado a partir de las estructuras creadas hasta el ultimo momento **/
  int q;
  q=cab;
  if(q!=-1){
    while(q!=-1){
      cout<<lista[q].dato<<endl;
      q=lista[q].sgte;
    }
```

```
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato){ /** Inserta al inicio de la lista **/
  int q;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo sgte apunta a NULL(-1) **/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=-1;
    *cab=q;
  }
  else{  /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el ultimo nodo-sgte apunten a NULL(-1) **/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=*cab;
    *cab=q;
    }
  cout<<dato<<" insertado al inicio de la lista "<<endl;
}


void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato){ /** Inserta al final de la lista **/
  int q,p;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a NULL(-1) **/
    nuevonodo(&q,lista,&(*cab1));
```

```
    lista[q].dato=dato;
    lista[q].sgte=-1;
    *cab=q;
  }
  else{ /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el ultimo nodo-sgte apunten a NULL(-1) **/
    nuevonodo(&q,lista,&(*cab1));
    p=*cab;
    lista[q].dato=dato;
    lista[q].sgte=-1;
    while(lista[p].sgte!=-1){
      p=lista[p].sgte;
    }
    lista[p].sgte=q;
  }
  cout<<dato<<" insertado al final de la lista "<<endl;
}


void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos){ /** Inserta en una posición dada y y reorganiza los elementos de la lista **/
  int i=1, cant=1 , p, q ,a ,b; /**Nodos auxiliares **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  if(pos==1){ /** Si la lista está vacia o se inserta a la 1° posición se llama a la función insertarinicio **/
    insertarinicio(lista, &(*cab), &(*cab1), dato);
  }
  else {
    while(lista[a].sgte!=-1){ /** Si la lista posee más de un elemento se halla la cantidad de nodos en la lista **/
      a=lista[a].sgte;
      cant++;
```

```
    }
    if(pos==cant+1){ /** Si se quiere insertar en la posición sgte. de toda la lista se llama a la función insertarfinal **/
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else if(pos<cant+1 && pos>1){
      lista[p].dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          i++;
        }
      lista[p].sgte=lista[b].sgte;
      lista[b].sgte=p;
    }
  }
  if(pos<=cant+1 && pos>=1){ /** Si la posición está entre los rangos normales de la lista se procede a enviar un mensaje que se insertó **/
    cout<<"Posicion insertada con exito"<<endl;
  }
  else{ /** En todo caso no realiza ninguna operación por estar fuera de los rangos de la lista **/
    cout<<"Posicion inexistente de la lista"<<endl;
  }
}


void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b ,c, d; /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
```

```
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[c].dato!=dato2 && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=lista[c].sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==1){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarinicio(lista, &(*cab), &(*cab1), dato);
    }
    else{
       if(pos<cant+1 && pos>1){
        lista[p].dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          i++;
        }
        lista[p].sgte=lista[b].sgte;
        lista[b].sgte=p;
       }
    }

    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
```

```
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}


void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){
/** Similar a la funcion insertarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b ,c, d; /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[c].dato!=dato2 && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=lista[c].sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){   /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarfinal(lista, &(*cab), &(*cab1), dato);
```

```
    }
    else{
      if(pos<=cant && pos>=1){
        d=lista[q].sgte;
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          d=lista[q].sgte;
          i++;
      }
        lista[q].sgte=p;
        lista[p].sgte=d;
      }
    }

    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}


void eliminarinicio(nodo lista[max], int *cab){ /** Elimina al inicio de la lista **/
  if(*cab!=-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if(lista[*cab].sgte==-1){ /** Si la lista es una sola, entoces apuntará a NULL **/
```

```
      *cab=-1;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo **/
      *cab=lista[*cab].sgte;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void eliminarfinal(nodo lista[max], int *cab){ /** Elimina al final de la lista **/
  int p,a;
  p=*cab;
  a=*cab;
  if(*cab!=-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if(lista[*cab].sgte==-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
      *cab=-1;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo **/
      while(lista[p].sgte!=-1){
        a=p;
        p=lista[p].sgte;
      }
      lista[a].sgte=-1;
    }
    cout<<"Se elimino el dato final de la lista"<<endl;
  }
  else{
    cout<<"No existe lista"<<endl;
```

```
  }
}


void eliminarposicion(nodo lista[max], int *cab, int pos){ /** Elimina una posición dada y reorganiza la lista **/
  int i=1, cant=1, p, q, a, b,c;
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){ /** Se halla la cantidad de nodos/datos de la lista **/
      a=lista[a].sgte;
      cant++;
    }
    if(pos==1){ /** Si la posicion a eliminar es inicial llamar a eliminarinicio **/
      eliminarinicio(lista,&(*cab));
    }
    else if(pos==cant){ /** Si la posicion es la posicion final de la lista llamar a eliminarfinal **/
        eliminarfinal(lista,&(*cab));
    }
    else if(pos<cant && pos>1) { /** Hallo un caso general para cualquier posicion que se encuentre entre la primera posición y ´ltima posición **/
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          c=lista[q].sgte;
          i++;
       }
      lista[b].sgte=c;
    }
    if(pos<=cant && pos>=1){ /** Mandar mensaje si se cumplió que la posición está dentro de los rangos de la lista **/
```

```
      cout<<"Posicion eliminada con exito"<<endl;
    }
    else{ /** En todo caso no se pudó realizar ninguna operación y mandar mensaje **/
      cout<<"Posicion inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }
}


void eliminardato(nodo lista[max], int *cab, int dato){ /** Similar a la funcion eliminarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b, c; /** Solo ubicaré la posición del dato y el resto lo haré como la función eliminarposicion **/
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      p=lista[p].sgte;
      pos++;
    }
    if(cant>=pos || pos==1){
      if(pos==1){           /** Este procedimiento a seguir es igual a la de la funcion eliminarposicion **/
        eliminarinicio(lista,&(*cab));
```

```
        }
        else {
          if(pos==cant){
            eliminarfinal(lista,&(*cab));
          }
          else{
            if(pos<cant && pos>1){
              while(i!=pos){
                b=q;
                q=lista[q].sgte;
                c=lista[q].sgte;
                i++;
              }
              lista[b].sgte=c;
            }
          }
        }
      }
      if(pos<=cant && pos>=1){ /** Mensajes de aviso si el dato existe o no **/
        cout<<dato<<" eliminado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }

}
```

void buscardato(nodo lista[max], int cab, int dato){ /** Busca una dato y avisa si se encuentra(con su posición) **/

```
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=-1){ /** Hallo la cantidad de nodos de la lista **/
      cant++;
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el
dato **/
      pos++;
      p=lista[p].sgte;
    }
  if(cant>=pos && pos>=1){ /**Mensaje de aviso si el dato se encuentra on no
dependiendo de la posicion ingresada **/
    cout<<dato<<" se encontro en la posicion "<<pos<<endl;
  }
  else{
    cout<<"Dato inexistente en la lista"<<endl;
  }
  }
}


void editardato(nodo lista[max], int cab, int dato, int dato2){ /** Similar a
buscardato, reemplaza el dato pedido por el dato anterior de la lista **/
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=-1){ /** Hallo la cantidad de nodos de la lista **/
      cant++;
```

```
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el dato **/
      pos++;
      p=lista[p].sgte;
    }
    if(cant>=pos){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      lista[p].dato=dato2; /** Aquí edito el dato último seleccionado de "p" **/
      cout<<dato<<" se encuentro en la posicion: "<<pos<< " y se edito por "<<dato2;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }

  }
}


void cantidad(nodo lista[max], int cab){ /** Determina la cantidad de elementos de la lista **/
  int p, cant=1;
  p=cab;
  if(cab!=-1){
    while(lista[p].sgte!=-1){ /** Cuenta los elementos yendo a través de cada dato **/
      p=lista[p].sgte;
      cant++;
    }
    cout<<"La lista posee: "<<cant<<" elementos"<<endl;
  }
  else{
```

```
    cout<<"La lista esta vacia"<<endl;
  }
}


void ordenardato(nodo lista[max], int cab){ /** Ordenamiento por el método de burbuja asociada a listas **/
  int t, p, q;
  p=cab;
  if(cab!=-1){
    while(lista[p].sgte!=-1){
      q=lista[p].sgte;
      while(q!=-1){
        if(lista[p].dato>lista[q].dato){
          t=lista[q].dato;
          lista[q].dato=lista[p].dato;
          lista[p].dato=t;
        }
        q=lista[q].sgte;
      }
      p=lista[p].sgte;
    }
    cout<<"Lista ordenada ascendentemente"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void guardar(nodo lista[max], int cab){ /** Guardar el archivo de la lista simple estática en forma de dato **/
  int p, dato;
  FILE *H;
```

```
  p=cab;
  if(p==-1){
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
  else{
    H=fopen("ListaSim.dat","w+");
    while(p!=-1){
      dato=lista[p].dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=lista[p].sgte;
    }
    fclose(H);
    cout<<"Archivo guardado"<<endl;
  }
}

void recuperar(nodo lista[max], int *cab, int *cab1){ /** Se recuperan los datos y se
insertar al final en una lista simple estática **/
  int p, dato;
  FILE *H;
  p=*cab;
  H=fopen("ListaSim.dat","r+");
  fread(&dato,sizeof(dato),1,H);
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
  while(!feof(H)){
    insertarfinal(lista,&(*cab),&(*cab1),dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

ANEXO 4.3.1

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

using namespace std;

struct nodo{ /** Declaro una estructura en forma de lista doble dinámica **/
  nodo *ant;
  int dato;
  nodo *sgte;
  };

void crearlista(nodo **lista); /** Funciones cabeceras **/
void mostrarlista(nodo *lista);
void insertarinicio(nodo **lista, int dato);
void insertarfinal(nodo **lista, int dato);
void insertarposicion(nodo **lista, int dato, int pos);
void insertarantesdato(nodo **lista, int dato, int dato2);
void insertardespuesdato(nodo **lista, int dato, int dato2);
void eliminarinicio(nodo **lista);
void eliminarfinal(nodo **lista);
void eliminarposicion(nodo **lista, int pos);
void eliminardato(nodo **lista, int dato);
void buscardato(nodo *lista, int dato);
void editardato(nodo **lista, int dato, int dato2);
void ordenardato(nodo *lista);
void cantidad(nodo *lista);
void guardar(nodo **lista);
void recuperar(nodo **lista);



int main(){
```

```
nodo *lista;
crearlista(&lista);
int opc, dato, dato2, pos;
do{
  system("color 1B");
  //system("cls");
  cout<<"\n\n\n";
  cout<<("*****MENU*****")<<endl; /** Por aquí se visualiza el MENU **/
  cout<<"1.  Mostrar lista"<<endl;
  cout<<"2.  Insertar al inicio"<<endl;
  cout<<"3.  Insertar al final"<<endl;
  cout<<"4.  Insertar en una posicion determinada"<<endl;
  cout<<"5.  Insertar antes de un dato"<<endl;
  cout<<"6.  Insertar despues de un dato"<<endl;
  cout<<"7.  Eliminar al inicio"<<endl;
  cout<<"8.  Eliminar al final"<<endl;
  cout<<"9.  Eliminar en una posicion determinada"<<endl;
  cout<<"10. Eliminar un dato"<<endl;
  cout<<"11. Buscar dato"<<endl;
  cout<<"12. Editar dato"<<endl;
  cout<<"13. Ordenar lista"<<endl;
  cout<<"14. Cantidad de datos"<<endl;
  cout<<"15. Guardar lista"<<endl;
  cout<<"16. Recuperar lista"<<endl;
  cout<<"17. Salir"<<endl;
  cout<<"Ingrese opcion: ";
  cin>>opc;
  cout<<"\n\n\n";
  switch(opc){
    case 1:  cout<<"LISTADO: "<<endl;
             mostrarlista(lista);
             break;
    case 2:  cout<<"Valor a insertar al inicio: ";
             cin>>dato;
```

```
            insertarinicio(&lista, dato);
            break;
    case 3:  cout<<"Valor a insertar al final: ";
            cin>>dato;
            insertarfinal(&lista, dato);
            break;
    case 4:  cout<<"Valor a insertar: ";
            cin>>dato;
            cout<<"En que posicion?: ";
            cin>>pos;
            insertarposicion(&lista, dato, pos);
            break;
    case 5:  cout<<"Dato a ingresar: ";
            cin>>dato;
            cout<<"Antes del dato: ";
            cin>>dato2;
            insertarantesdato(&lista, dato, dato2);
            break;
    case 6:  cout<<"Dato a ingresar: ";
            cin>>dato;
            cout<<"Despues del dato: ";
            cin>>dato2;
            insertardespuesdato(&lista, dato, dato2);
            break;
    case 7:  eliminarinicio(&lista);
            break;
    case 8:  eliminarfinal(&lista);
            break;
    case 9:  cout<<"Posicion a eliminar: ";
            cin>>pos;
            eliminarposicion(&lista, pos);
            break;
    case 10: cout<<"Dato a eliminar: ";
            cin>>dato;
```

```
                eliminardato(&lista, dato);
                break;
        case 11: cout<<"Dato a buscar: ";
                cin>>dato;
                buscardato(lista, dato);
                break;
        case 12: cout<<"Dato a buscar: ";
                cin>>dato;
                cout<<"Editar por: ";
                cin>>dato2;
                editardato(&lista, dato, dato2);
                break;
        case 13: ordenardato(lista);
                break;
        case 14: cantidad(lista);
                break;
        case 15: guardar(&lista);
                break;
        case 16: recuperar(&lista);
                break;
      }
   }
   while(opc!=17);
   system("pause");
   return(0);
}



void crearlista(nodo **lista){
   *lista=NULL;
}
```

```
void mostrarlista(nodo *lista){ /** Muestra cada nodo visualizado a partir de las estructuras creadas hasta el ultimo momento **/
  nodo *q;
  q=lista;
  if(q!=NULL){
    while(q!=NULL){ /** Recorre todos los nodos **/
      cout<<q->dato<<endl;
      q=q->sgte;
    }
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void insertarinicio(nodo **lista, int dato){ /** Se inserta dato al inicio de la lista **/
    nodo *p; /** Nodos auxiliares **/
    p=new struct nodo;
    p->dato=dato;
    (*lista)->ant=p;
    p->sgte=*lista;
    p->ant=NULL;
    *lista=p; /** Al final la cabeza lista apuntará al nodo creado con el dato ingresado **/
    cout<<"Se inserto al inicio de la lista: "<<dato<<endl;
}


void insertarfinal(nodo **lista, int dato){
    nodo *p;
    nodo *q;
    q=new struct nodo;
```

```
    if(*lista==NULL){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a NULL(-1) **/
      q->dato=dato;
      q->ant=NULL;
      q->sgte=NULL;
      *lista=q;
    }
    else{ /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el ultimo nodo-sgte apunten a NULL(-1) **/
      p=*lista;
      q->dato=dato;
      q->sgte=NULL;
      while(p->sgte!=NULL){
        p=p->sgte;
      }
      q->ant=p;
      p->sgte=q;
    }
    cout<<dato<<" insertado al final de la lista "<<endl;
}


void insertarposicion(nodo **lista, int dato, int pos){
  nodo *p; /**Nodos auxiliares **/
  nodo *q;
  nodo *a;
  nodo *b;
  p=new struct nodo;
  int i=1, cant=1, aux;
  q=*lista;
  a=*lista;
  if(pos==1){ /** Si la lista está vacia o se inserta a la 1° posición se llama a la función insertarinicio **/
    insertarinicio(&(*lista),dato);
```

```
  }
  else {
    while(a->sgte!=NULL){ /** Si la lista posee más de un elemento se halla la cantidad de nodos en la lista **/
      a=a->sgte;
      cant++;
    }
    if(pos==cant+1){ /** Si se quiere insertar en la posición sgte. de toda la lista se llama a la función insertarfinal **/
      insertarfinal(&(*lista),dato);
    }
    else if(pos<cant+1 && pos>1){
      p->dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
      if(pos<=cant && pos>1){
        while(i!=pos){
          b=q;
          q=q->sgte;
          i++;
        }
        p->sgte=b->sgte;
        p->ant=b;
        b->sgte=p;
        q->ant=p;
      }
    }
  }
  if(pos<=cant+1 && pos>=1){ /** Si la posición está entre los rangos normales de la lista se procede a enviar un mensaje que se insertó **/
    cout<<"Posicion insertada con exito"<<endl;
  }
  else{ /** En todo caso no realiza ninguna operación por estar fuera de los rangos de la lista **/
    cout<<"Posicion inexistente de la lista"<<endl;
```

```
  }
}


void insertarantesdato(nodo **lista, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  nodo *p;                          /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  nodo *d;
  p=new struct nodo;
  p->dato=dato;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=c->sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){
      if(c->dato!=dato2){
        pos=cant+1;
      }
    }
```

```
    if(pos==1){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarinicio(&(*lista),dato);
    }
    else{
      if(pos<=cant && pos>=1){
        while(i!=pos){
          b=q;
          q=q->sgte;
          i++;
        }
        p->sgte=b->sgte;
        p->ant=b;
        b->sgte=p;
        q->ant=p;
      }
    }

    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}
```

```
void insertardespuesdato(nodo **lista, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  nodo *p;                          /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  nodo *d;
  p=new struct nodo;
  p->dato=dato;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=c->sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){
      if(c->dato!=dato2){
        pos=cant+1;
      }
    }
    if(pos==cant){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarfinal(&(*lista),dato);
    }
```

```
    else{
      if(pos<=cant && pos>=1){
        d=q->sgte;
        while(i!=pos){
          b=q;
          q=q->sgte;
          d=q->sgte;
          i++;
      }
        q->sgte=p;
        p->ant=q;
        p->sgte=d;
        d->ant=p;
      }
    }

    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}
```

```
void eliminarinicio(nodo **lista){
  if(*lista!=NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if((*lista)->sgte==NULL){ /** Si la lista es una sola, entoces apuntará a NULL **/
      *lista=NULL;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo **/
      *lista=(*lista)->sgte;
      (*lista)->ant=NULL;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void eliminarfinal(nodo **lista){
  nodo *p;
  nodo *a;
  p=*lista;
  a=*lista;
  if(*lista!=NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if((*lista)->sgte==NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
      *lista=NULL;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo **/
      while(p->sgte!=NULL){
        a=p;
        p=p->sgte;
```

```
        }
        a->sgte=NULL;
      }
      cout<<"Se elimino el dato final de la lista"<<endl;
    }
    else{
      cout<<"No existe lista"<<endl;
    }
}


void eliminarposicion(nodo **lista, int pos){
    nodo *q;
    nodo *a;
    nodo *b;
    nodo *c;
    int i=1, cant=1;
    q=*lista;
    a=*lista;
    if(*lista!=NULL){
      while(a->sgte!=NULL){ /** Se halla la cantidad de nodos/datos de la lista **/
        a=a->sgte;
        cant++;
      }
      if(pos==1){ /** Si la posicion a eliminar es inicial llamar a eliminarinicio **/
        eliminarinicio(&(*lista));
      }
      else if(pos==cant){ /** Si la posicion es la posicion final de la lista llamar a
eliminarfinal **/
          eliminarfinal(&(*lista));
      }
      else if(pos<cant && pos>1) { /** Hallo un caso general para cualquier posicion
que se encuentre entre la primera posición y última posición **/
          if(pos<=cant && pos>1){
```

```
            while(i!=pos){
              b=q;
              q=q->sgte;
              c=q->sgte;
              i++;
            }
            b->sgte=c;
            c->ant=b;
          }
        }
        if(pos<=cant && pos>=1){ /** Mandar mensaje si se cumplió que la posición está dentro de los rangos de la lista **/
          cout<<"Posicion eliminada con exito"<<endl;
        }
        else{ /** En todo caso no se pudó realizar ninguna operación y mandar mensaje **/
          cout<<"Posicion inexistente de la lista"<<endl;
        }
      }
      else{
        cout<<"Lista vacia!"<<endl;
      }
}


void eliminardato(nodo **lista, int dato){ /** Similar a la funcion eliminarposicion **/
  nodo *p;                          /** Solo ubicaré la posición del dato y el resto lo haré como la función eliminarposicion **/
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  int i=1, cant=1, pos=1;
```

```
  q=*lista;
  a=*lista;
  p=*lista;
  if(*lista!=NULL){
    while(a->sgte!=NULL){
      a=a->sgte;
      cant++;
    }
    while(p->dato!=dato && p->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      p=p->sgte;
      pos++;
    }
    if(pos==cant){
      if(p->dato!=dato){
        pos=cant+1;
      }
    }
    if(cant>=pos || pos==1){
      if(pos==1){          /** Este procedimiento a seguir es igual a la de la funcion eliminarposicion **/
        eliminarinicio(&(*lista));
      }
      else {
        if(pos==cant){
          eliminarfinal(&(*lista));
        }
        else{
          if(pos<cant && pos>1){
            while(i!=pos){
              b=q;
              q=q->sgte;
              c=q->sgte;
              i++;
```

```
            }
              b->sgte=c;
              c->ant=b;
            }
          }
        }
      }
      if(pos<=cant && pos>=1){ /** Mensajes de aviso si el dato existe o no **/
        cout<<dato<<" eliminado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }

}



void buscardato(nodo *lista, int dato){
    nodo *p;
    nodo *q;
    int i=1, a=1;
    if(lista!=NULL){
      q=lista;
      while(q->sgte!=NULL){ /** Hallo la cantidad de nodos de la lista **/
        q=q->sgte;
        a++;
      }
      p=lista;
```

```
    while(p->dato!=dato && p->sgte!=NULL && a>=i){ /** Realizo operaciones para buscar el dato **/
      p=p->sgte;
      i++;
    }
    if(i==a){ /** Si el nodo p llega al último dato **/
      if(p->dato!=dato){
        i=a+1;
      }
    }
    if(a>=i){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      cout<<dato<<" se encuentra en la posicion: "<<i;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}


void editardato(nodo **lista, int dato, int dato2){
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if((*lista)!=NULL){
    q=*lista;
    while(q->sgte!=NULL){ /** Hallo la cantidad de nodos de la lista **/
      q=q->sgte;
      a++;
    }
```

```
    p=*lista;
    while(p->dato!=dato && p->sgte!=NULL && a>=i){ /** Realizo operaciones para buscar el dato **/
      p=p->sgte;
      i++;
    }
    if(i==a){ /** Si el nodo p llega al último dato **/
      if(p->dato!=dato){
        i=a+1;
      }
    }
    if(p->dato==dato){ /** Si el p->dato es igual al dato encontrado entonces se modifica **/
      p->dato=dato2;
    }
    if(a>=i){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      cout<<dato<<" se encuentro en la posicion: "<<i<< "y se edito por "<<dato2;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}


void ordenardato(nodo *lista){ /** Ordenamiento por el método de burbuja asociada a listas **/
  int t;
  nodo *p;
  nodo *q;
```

```
  p=lista;
  if(lista!=NULL){
    while(p->sgte!=NULL){
      q=p->sgte;
      while(q!=NULL){
        if(p->dato>q->dato){
          t=q->dato;
          q->dato=p->dato;
          p->dato=t;
        }
        q=q->sgte;
      }
      p=p->sgte;
    }
    cout<<"Lista ordenada ascendentemente"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void cantidad(nodo *lista){
  int cant=1;
  nodo *p;
  p=lista;
  if(lista!=NULL){
    while(p->sgte!=NULL){ /** Cuenta los elementos yendo a través de cada dato
**/
      p=p->sgte;
      cant++;
    }
    cout<<"La lista posee: "<<cant<<" elementos"<<endl;
  }
```

```
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}


void guardar(nodo **lista){ /** Guardar el archivo de la lista doble en forma de dato
**/
  int dato;
  FILE *H;
  nodo *p;
  p=*lista;
  if(p==NULL){
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
  else{
    H=fopen("ListaDob2.dat","w+");
    while(p!=NULL){
      dato=p->dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=p->sgte;
    }
    fclose(H);
    cout<<"Archivo guardado"<<endl;
  }
}

void recuperar(nodo **lista){ /** Se recuperan los datos y se insertar al final en una
lista doble **/
  FILE *H;
  int dato;
  H=fopen("ListaDob2.dat","r+");
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
```

```
    return;
  }
  fread(&dato,sizeof(dato),1,H);
  while(!feof(H)){
    insertarfinal(&(*lista), dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

ANEXO 4.3.2

4.3.2 Vista panorámica del menú de las listas dobles estáticas:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define max 100 /** Define la cantidad máxima de listas estáticas con la que voy a trabajar **/

using namespace std;

struct nodo{ /** Declaro una estructura en forma de lista doble estática **/
  int ant;
  int dato;
  int sgte;
};

void crear(nodo lista[max], int *cab, int *cab1); /** Funciones cabeceras **/
void nuevonodo(int *p, nodo lista[max], int *cab1);
void mostrarlista(nodo lista[max], int cab);
void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato);
void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato);
void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos);
void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2);
void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2);
void eliminarinicio(nodo lista[max], int *cab);
void eliminarfinal(nodo lista[max], int *cab);
void eliminarposicion(nodo lista[max], int *cab, int pos);
void eliminardato(nodo lista[max], int *cab, int dato);
void buscardato(nodo lista[max], int cab, int dato);
void editardato(nodo lista[max], int cab, int dato, int dato2);
void ordenardato(nodo lista[max], int cab);
void cantidad(nodo lista[max], int cab);
void guardar(nodo lista[max], int cab);
```

```
void recuperar(nodo lista[max], int *cab, int *cab1);


int main(){
  nodo lista[max]; /** Declaro una variable de la lista estática simple **/
  int cab, cab1, opc, dato, dato2, pos; /** La cabecera de la lista será una variable cab
**/
  crear(lista,&cab,&cab1); /** cab1 ayudará a la creación de nuevos nodos **/
  do{
    system("color 1B");
    cout<<"\n\n\n";
    cout<<("*****MENU*****")<<endl; /** Por aquí se visualiza el MENU **/
    cout<<"1.  Mostrar lista"<<endl;
    cout<<"2.  Insertar al inicio"<<endl;
    cout<<"3.  Insertar al final"<<endl;
    cout<<"4.  Insertar en una posicion determinada"<<endl;
    cout<<"5.  Insertar antes de un dato"<<endl;
    cout<<"6.  Insertar despues de un dato"<<endl;
    cout<<"7.  Eliminar al inicio"<<endl;
    cout<<"8.  Eliminar al final"<<endl;
    cout<<"9.  Eliminar en una posicion determinada"<<endl;
    cout<<"10. Eliminar un dato"<<endl;
    cout<<"11. Buscar dato"<<endl;
    cout<<"12. Editar dato"<<endl;
    cout<<"13. Ordenar lista"<<endl;
    cout<<"14. Numero de elementos de la lista"<<endl;
    cout<<"15. Guardar lista"<<endl;
    cout<<"16. Recuperar lista"<<endl;
    cout<<"17. Salir"<<endl;
    cout<<"Ingrese opcion: ";
    cin>>opc;
    cout<<"\n\n\n";
    switch(opc){
      case 1:  cout<<"LISTADO: "<<endl;
```

```
        mostrarlista(lista,cab);
        break;
case 2:  cout<<"Valor a insertar al inicio: ";
        cin>>dato;
        insertarinicio(lista,&cab,&cab1,dato);
        break;
case 3:  cout<<"Valor a insertar al final: ";
        cin>>dato;
        insertarfinal(lista,&cab,&cab1,dato);
        break;
case 4:  cout<<"Valor a insertar: ";
        cin>>dato;
        cout<<"En que posicion?: ";
        cin>>pos;
        insertarposicion(lista,&cab,&cab1,dato,pos);
        break;
case 5:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Antes del dato: ";
        cin>>dato2;
        insertarantesdato(lista,&cab,&cab1,dato,dato2);
        break;
case 6:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Despues del dato: ";
        cin>>dato2;
        insertardespuesdato(lista,&cab,&cab1,dato,dato2);
        break;
case 7:  eliminarinicio(lista,&cab);
        break;
case 8:  eliminarfinal(lista,&cab);
        break;
case 9:  cout<<"Posicion a eliminar: ";
        cin>>pos;
```

```
                eliminarposicion(lista,&cab,pos);
                break;
        case 10:  cout<<"Dato a eliminar: ";
                cin>>dato;
                eliminardato(lista,&cab,dato);
                break;
        case 11: cout<<"Dato a buscar: ";
                cin>>dato;
                buscardato(lista,cab,dato);
                break;
        case 12: cout<<"Dato a buscar: ";
                cin>>dato;
                cout<<"Editar por: ";
                cin>>dato2;
                editardato(lista, cab, dato, dato2);
                break;
        case 13: ordenardato(lista,cab);
                break;
        case 14: cantidad(lista, cab);
                break;
        case 15: guardar(lista,cab);
                break;
        case 16: recuperar(lista,&cab,&cab1);
                break;
      }
   }
   while(opc!=17);
   system("pause");
   return(0);
}
```

```
void crear(nodo lista[max], int *cab, int *cab1){   /** Crea todas las estructuras posibles con el límite máximo declarado simulando la creación de espacio de nodos **/
  int i;
  *cab=-1;
  *cab1 =0;
  lista[0].ant =-1;
  lista[0].sgte =1;
  for(i=1;i<max-1;i++)
  {
    lista[i].dato = 0;
    lista[i].ant=i-1;
    lista[i].sgte = i+1;
  }
  lista[max-1].sgte =-1;
  lista[max-1].ant =max-2;
}

void nuevonodo(int *p, nodo lista[max], int *cab1) /** Crea un nuevo nodo a partir de la posicion de cab1 y la función crear **/
{
  *p = *cab1;
  if(*cab1 == -1)
  {
    printf(" No hay suficiente memoria\n\n");
    exit(1);
  }
  *cab1 = lista[*cab1].sgte;
}


void mostrarlista(nodo lista[max], int cab){ /** Muestra cada nodo visualizado a partir de las estructuras creadas hasta el ultimo momento **/
  int q;
```

```
  q=cab;
  if(q!=-1){
    while(q!=-1){
      cout<<lista[q].dato<<endl;
      q=lista[q].sgte;
    }
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

```
void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato){ /** Inserta al inicio de la lista **/
  int q;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a NULL(-1) **/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=-1;
    lista[q].ant=-1;
    *cab=q;
  }
  else{  /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el primer nodo-ant y el ultimo nodo-sgte apunten a NULL(-1) **/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[*cab].ant=q;
    lista[q].sgte=*cab;
    lista[q].ant=-1;
    *cab=q;
    }
  cout<<dato<<" insertado al inicio de la lista "<<endl;
```

```
}


void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato){ /** Inserta al final de la lista **/
  int q,p;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a NULL(-1) **/
     nuevonodo(&q,lista,&(*cab1));
     lista[q].dato=dato;
     lista[q].sgte=-1;
     lista[q].ant=-1;
     *cab=q;
  }
  else{ /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el primer nodo-ant y el ultimo nodo-sgte apunten a NULL(-1) **/
     nuevonodo(&q,lista,&(*cab1));
     p=*cab;
     lista[q].dato=dato;
     lista[q].sgte=-1;
     while(lista[p].sgte!=-1){
       p=lista[p].sgte;
     }
     lista[q].ant=p;
     lista[p].sgte=q;
  }
  cout<<dato<<" insertado al final de la lista "<<endl;
}


void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos){ /** Inserta en una posición dada y y reorganiza los elementos de la lista **/
  int i=1, cant=1 , p, q ,a ,b; /**Nodos auxiliares **/
  nuevonodo(&p,lista,&(*cab1));
```

```
  q=*cab;
  a=*cab;
  if(pos==1){ /** Si la lista está vacia o se inserta a la 1° posición se llama a la función insertarinicio **/
    insertarinicio(lista, &(*cab), &(*cab1), dato);
  }
  else {
    while(lista[a].sgte!=-1){ /** Si la lista posee más de un elemento se halla la cantidad de nodos en la lista **/
      a=lista[a].sgte;
      cant++;
    }
    if(pos==cant+1){ /** Si se quiere insertar en la posición sgte. de toda la lista se llama a la función insertarfinal **/
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else if(pos<cant+1 && pos>1){
      lista[p].dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
      if(pos<=cant && pos>1){
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          i++;
        }
      lista[p].sgte=lista[b].sgte;
      lista[p].ant=b;
      lista[b].sgte=p;
      lista[q].ant=p;
      }
    }
  }
  if(pos<=cant+1 && pos>=1){ /** Si la posición está entre los rangos normales de la lista se procede a enviar un mensaje que se insertó **/
```

```
    cout<<"Posicion insertada con exito"<<endl;
  }
  else{ /** En todo caso no realiza ninguna operación por estar fuera de los rangos de la lista **/
    cout<<"Posicion inexistente de la lista"<<endl;
  }
}


void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b ,c, d; /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[c].dato!=dato2 && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=lista[c].sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==1){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarinicio(lista, &(*cab), &(*cab1), dato);
    }
    else{
       if(pos<cant+1 && pos>1){
```

```
        lista[p].dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
        if(pos<=cant && pos>1){
          while(i!=pos){
            b=q;
            q=lista[q].sgte;
            i++;
          }
          lista[p].sgte=lista[b].sgte;
          lista[p].ant=b;
          lista[b].sgte=p;
          lista[q].ant=p;
        }
        }
    }
    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}


void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b ,c, d; /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
```

```
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[c].dato!=dato2 && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=lista[c].sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){   /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else{
      if(pos<=cant && pos>=1){
        d=lista[q].sgte;
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          d=lista[q].sgte;
          i++;
        }
        lista[q].sgte=p;
        lista[p].ant=q;
        lista[p].sgte=d;
        lista[d].ant=p;
      }
    }
```

```
    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}


void eliminarinicio(nodo lista[max], int *cab){ /** Elimina al inicio de la lista **/
  if(*cab!=-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if(lista[*cab].sgte==-1){ /** Si la lista es una sola, entoces apuntará a NULL **/
      *cab=-1;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo **/
      *cab=lista[*cab].sgte;
      lista[*cab].ant=-1;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void eliminarfinal(nodo lista[max], int *cab){ /** Elimina al final de la lista **/
```

```
  int p,a;
  p=*cab;
  a=*cab;
  if(*cab!=-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if(lista[*cab].sgte==-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
      *cab=-1;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo **/
      while(lista[p].sgte!=-1){
        a=p;
        p=lista[p].sgte;
      }
      lista[a].sgte=-1;
    }
    cout<<"Se elimino el dato final de la lista"<<endl;
  }
  else{
    cout<<"No existe lista"<<endl;
  }
}


void eliminarposicion(nodo lista[max], int *cab, int pos){ /** Elimina una posición dada y reorganiza la lista **/
  int i=1, cant=1, p, q, a, b,c;
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){ /** Se halla la cantidad de nodos/datos de la lista **/
      a=lista[a].sgte;
      cant++;
```

```
    }
    if(pos==1){ /** Si la posicion a eliminar es inicial llamar a eliminarinicio **/
      eliminarinicio(lista,&(*cab));
    }
    else if(pos==cant){ /** Si la posicion es la posicion final de la lista llamar a eliminarfinal **/
        eliminarfinal(lista,&(*cab));
    }
    else if(pos<cant && pos>1) { /** Hallo un caso general para cualquier posicion que se encuentre entre la primera posición y ´ltima posición **/
         while(i!=pos){
          b=q;
          q=lista[q].sgte;
          c=lista[q].sgte;
          i++;
       }
       lista[b].sgte=c;
       lista[c].ant=b;
    }
    if(pos<=cant && pos>=1){ /** Mandar mensaje si se cumplió que la posición está dentro de los rangos de la lista **/
       cout<<"Posicion eliminada con exito"<<endl;
    }
    else{ /** En todo caso no se pudó realizar ninguna operación y mandar mensaje **/
       cout<<"Posicion inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }
}
```

```
void eliminardato(nodo lista[max], int *cab, int dato){ /** Similar a la funcion eliminarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b, c; /** Solo ubicaré la posición del dato y el resto lo haré como la función eliminarposicion **/
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=-1){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      p=lista[p].sgte;
      pos++;
    }
    if(cant>=pos || pos==1){
      if(pos==1){           /** Este procedimiento a seguir es igual a la de la funcion eliminarposicion **/
        eliminarinicio(lista,&(*cab));
      }
      else {
        if(pos==cant){
          eliminarfinal(lista,&(*cab));
        }
        else{
          if(pos<cant && pos>1){
            while(i!=pos){
              b=q;
              q=lista[q].sgte;
              c=lista[q].sgte;
              i++;
            }
```

```
                lista[b].sgte=c;
                lista[c].ant=b;
              }
            }
          }
      }
      if(pos<=cant && pos>=1){ /** Mensajes de aviso si el dato existe o no **/
        cout<<dato<<" eliminado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }

}


void buscardato(nodo lista[max], int cab, int dato){ /** Busca una dato y avisa si se encuentra(con su posición) **/
    int p=cab, q=cab, cant=1, pos=1;
    if(cab==-1){
      printf("Lista Vacia");
    }
    else{
      while(lista[q].sgte!=-1){ /** Hallo la cantidad de nodos de la lista **/
        cant++;
        q=lista[q].sgte;
      }
      while(lista[p].dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el dato **/
        pos++;
```

```
      p=lista[p].sgte;
    }
  if(cant>=pos && pos>=1){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
    cout<<dato<<" se encontro en la posicion "<<pos<<endl;
  }
  else{
    cout<<"Dato inexistente en la lista"<<endl;
  }
  }
}


void editardato(nodo lista[max], int cab, int dato, int dato2){ /** Similar a buscardato, reemplaza el dato pedido por el dato anterior de la lista **/
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=-1){ /** Hallo la cantidad de nodos de la lista **/
      cant++;
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el dato **/
      pos++;
      p=lista[p].sgte;
    }
    if(cant>=pos){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      lista[p].dato=dato2; /** Aquí edito el dato último seleccionado de "p" **/
      cout<<dato<<" se encuentro en la posicion: "<<pos<< " y se edito por "<<dato2;
```

```
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }

  }
}


void cantidad(nodo lista[max], int cab){ /** Determina la cantidad de elementos de la lista **/
  int p, cant=1;
  p=cab;
  if(cab!=-1){
    while(lista[p].sgte!=-1){ /** Cuenta los elementos yendo a través de cada dato **/
      p=lista[p].sgte;
      cant++;
    }
    cout<<"La lista posee: "<<cant<<" elementos"<<endl;
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}


void ordenardato(nodo lista[max], int cab){ /** Ordenamiento por el método de burbuja asociada a listas **/
  int t, p, q;
  p=cab;
  if(cab!=-1){
    while(lista[p].sgte!=-1){
      q=lista[p].sgte;
```

```
      while(q!=-1){
        if(lista[p].dato>lista[q].dato){
          t=lista[q].dato;
          lista[q].dato=lista[p].dato;
          lista[p].dato=t;
        }
        q=lista[q].sgte;
      }
      p=lista[p].sgte;
    }
    cout<<"Lista ordenada ascendentemente"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

void guardar(nodo lista[max], int cab){ /** Guardar el archivo de la lista simple estática en forma de dato **/

```
  int p, dato;
  FILE *H;
  p=cab;
  if(p==-1){
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
  else{
    H=fopen("ListaDob.dat","w+");
    while(p!=-1){
      dato=lista[p].dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=lista[p].sgte;
    }
    fclose(H);
```

```
    cout<<"Archivo guardado"<<endl;
  }
}


void recuperar(nodo lista[max], int *cab, int *cab1){ /** Se recuperan los datos y se insertar al final en una lista simple estática **/
  int p, dato;
  FILE *H;
  p=*cab;
  H=fopen("ListaDob.dat","r+");
  fread(&dato,sizeof(dato),1,H);
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
  while(!feof(H)){
    insertarfinal(lista,&(*cab),&(*cab1),dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

ANEXO 4.3.3

4.3.3 Vista panorámica del menú de las listas simples circulares dinámicas:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

using namespace std;

struct nodo{ /** Declaro una estructura en forma de lista simple circular dinámica**/
  int dato;
  nodo *sgte;
  };

void crearlista(nodo **lista); /** Funciones cabeceras **/
void mostrarlista(nodo *lista);
void insertarinicio(nodo **lista, int dato);
void insertarfinal(nodo **lista, int dato);
void insertarposicion(nodo **lista, int dato, int pos);
void insertarantesdato(nodo **lista, int dato, int dato2);
void insertardespuesdato(nodo **lista, int dato, int dato2);
void eliminarinicio(nodo **lista);
void eliminarfinal(nodo **lista);
void eliminarposicion(nodo **lista, int pos);
void eliminardato(nodo **lista, int dato);
void buscardato(nodo *lista, int dato);
void editardato(nodo **lista, int dato, int dato2);
void ordenardato(nodo *lista);
void cantidad(nodo *lista);
void guardar(nodo **lista);
void recuperar(nodo **lista);



int main(){
```

```
nodo *lista;
crearlista(&lista);
int opc, dato, dato2, pos;
do{
  system("color 1B");
  //system("cls");
  cout<<"\n\n\n";
  cout<<("*****MENU*****")<<endl; /** Por aquí se visualiza el MENU **/
  cout<<"1.  Mostrar lista"<<endl;
  cout<<"2.  Insertar al inicio"<<endl;
  cout<<"3.  Insertar al final"<<endl;
  cout<<"4.  Insertar en una posicion determinada"<<endl;
  cout<<"5.  Insertar antes de un dato"<<endl;
  cout<<"6.  Insertar despues de un dato"<<endl;
  cout<<"7.  Eliminar al inicio"<<endl;
  cout<<"8.  Eliminar al final"<<endl;
  cout<<"9.  Eliminar en una posicion determinada"<<endl;
  cout<<"10. Eliminar un dato"<<endl;
  cout<<"11. Buscar dato"<<endl;
  cout<<"12. Editar dato"<<endl;
  cout<<"13. Ordenar lista"<<endl;
  cout<<"14. Cantidad de datos"<<endl;
  cout<<"15. Guardar lista"<<endl;
  cout<<"16. Recuperar lista"<<endl;
  cout<<"17. Salir"<<endl;
  cout<<"Ingrese opcion: ";
  cin>>opc;
  cout<<"\n\n\n";
  switch(opc){
    case 1:  cout<<"LISTADO: "<<endl;
             mostrarlista(lista);
             break;
    case 2:  cout<<"Valor a insertar al inicio: ";
             cin>>dato;
```

```
        insertarinicio(&lista, dato);
        break;
case 3:  cout<<"Valor a insertar al final: ";
        cin>>dato;
        insertarfinal(&lista, dato);
        break;
case 4:  cout<<"Valor a insertar: ";
        cin>>dato;
        cout<<"En que posicion?: ";
        cin>>pos;
        insertarposicion(&lista, dato, pos);
        break;
case 5:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Antes del dato: ";
        cin>>dato2;
        insertarantesdato(&lista, dato, dato2);
        break;
case 6:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Despues del dato: ";
        cin>>dato2;
        insertardespuesdato(&lista, dato, dato2);
        break;
case 7:  eliminarinicio(&lista);
        break;
case 8:  eliminarfinal(&lista);
        break;
case 9:  cout<<"Posicion a eliminar: ";
        cin>>pos;
        eliminarposicion(&lista, pos);
        break;
case 10: cout<<"Dato a eliminar: ";
        cin>>dato;
```

```
                eliminardato(&lista, dato);
                break;
        case 11: cout<<"Dato a buscar: ";
                cin>>dato;
                buscardato(lista, dato);
                break;
        case 12: cout<<"Dato a buscar: ";
                cin>>dato;
                cout<<"Editar por: ";
                cin>>dato2;
                editardato(&lista, dato, dato2);
                break;
        case 13: ordenardato(lista);
                break;
        case 14: cantidad(lista);
                break;
        case 15: guardar(&lista);
                break;
        case 16: recuperar(&lista);
                break;
      }
   }
   while(opc!=17);
   system("pause");
   return(0);
}



void crearlista(nodo **lista){
   *lista=NULL;
}
```

```
void mostrarlista(nodo *lista){
  nodo *p; /** Declaro un nodo p para recorrer la lista */
  if(lista!=NULL){ /** Pregunta si la lista está vacía */
    p=lista; /** El nodo p apunta a la lista */
    cout<<p->dato<<endl; /** Leo el primer dato */
    p=p->sgte;  /** Apunto al siguiente nodo */
    while(p!=lista){ /** Leo los demás datos comenzando a partir del 2° dato hasta que llegue a la lista */
      cout<<p->dato<<endl;
      p=p->sgte;
    }
    cout<<endl;
  }
  else{ /** Si la lista está vacía retorna mensaje de aviso */
    cout<<"No existe lista"<<endl;
  }
}


void insertarinicio(nodo **lista, int dato){
    nodo *p; /** Nodos auxiliares **/
    nodo *q;
    p=new struct nodo;
    p->dato=dato;
    if(*lista==NULL){ /** Si la lista esta vacia se crea uno nuevo, enlazandose a sí mismo**/
      p->sgte=p;
      *lista=p;
    }
    else{ /** Si la lista contiene uno o más elementos se agregan, teniendo en cuenta que el primer y último nodo se enlazan **/
      q=*lista;
      while(q->sgte!=*lista){
        q=q->sgte;
```

```
        }
        q->sgte=p;
        p->sgte=*lista;
        *lista=p;
    }
    cout<<"Se inserto al inicio de la lista: "<<dato<<endl;
}


void insertarfinal(nodo **lista, int dato){
    nodo *p; /**Nodos auxiliares **/
    nodo *q;
    p=new struct nodo;
    p->dato=dato;
    if((*lista)==NULL){ /** Si la lista esta vacia se crea uno nuevo, enlazandose a sí mismo**/
        p->sgte=p;
        p->sgte=p;
        *lista=p;
    }
    else{ /** Si la lista contiene uno o más elementos se agregan, teniendo en cuenta que el primer y último nodo se enlazan **/
        q=*lista;
        while(q->sgte!=(*lista)){
          q=q->sgte;
        }
        q->sgte=p;
        p->sgte=*lista;
    }
    cout<<"Se inserto al final de la lista: "<<dato<<endl;
}


void insertarposicion(nodo **lista, int dato, int pos){
```

```
  nodo *p; /**Nodos auxiliares **/
  nodo *q;
  nodo *a;
  nodo *b;
  p=new struct nodo;
  int i=1, cant=1, aux;
  q=*lista;
  a=*lista;
  if(pos==1){ /** Si la lista está vacia o se inserta a la 1° posición se llama a la función insertarinicio **/
    insertarinicio(&(*lista),dato);
  }
  else {
    while(a->sgte!=(*lista)){ /** Si la lista posee más de un elemento se halla la cantidad de nodos en la lista **/
      a=a->sgte;
      cant++;
    }
    if(pos==cant+1){ /** Si se quiere insertar en la posición sgte. de toda la lista se llama a la función insertarfinal **/
      insertarfinal(&(*lista),dato);
    }
    else if(pos<cant+1 && pos>1){
      p->dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
      while(i!=pos){
        b=q;
        q=q->sgte;
        i++;
      }
      p->sgte=b->sgte;
      b->sgte=p;
    }
  }
```

```
  if(pos<=cant+1 && pos>=1){ /** Si la posición está entre los rangos normales de la lista se procede a enviar un mensaje que se insertó **/
    cout<<"Posicion insertada con exito"<<endl;
  }
  else{ /** En todo caso no realiza ninguna operación por estar fuera de los rangos de la lista **/
    cout<<"Posicion inexistente de la lista"<<endl;
  }
}


void insertarantesdato(nodo **lista, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  nodo *p;                          /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  p=new struct nodo;
  p->dato=dato;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=(*lista)){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2  && cant>=pos){ /** Realizo operaciones para buscar el dato **/
      c=c->sgte;
      pos++;
```

```
    }
    if(pos==1){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarinicio(&(*lista),dato);
    }
    else{
      if(pos<=cant && pos>=1){
        while(i!=pos){
        b=q;
        q=q->sgte;
        i++;
      }
      p->sgte=b->sgte;
      b->sgte=p;
      }
    }

    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}
```

```
void insertardespuesdato(nodo **lista, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  nodo *p;                          /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  p=new struct nodo;
  p->dato=dato;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=(*lista)){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2  && cant>=pos){ /** Realizo operaciones para buscar el dato **/
      c=c->sgte;
      pos++;
    }
    if(pos==cant){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarfinal(&(*lista),dato);
    }
    else{
      if(pos<=cant && pos>=1){
        while(i!=pos){
          b=q;
          q=q->sgte;
          i++;
```

```
        }
          p->sgte=q->sgte;
          q->sgte=p;
        }
      }

      if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
        cout<<dato<<" insertado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }

}


void eliminarinicio(nodo **lista){
  nodo *p;
  if(*lista!=NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
      if(*lista==(*lista)->sgte){ /** Si la lista es una sola, entoces apuntará a NULL **/
        *lista=NULL;
      }
      else{    /** En todo caso, se realizará el proceso de eliminar al inicio, siempre enlazando el primero con el final **/
        p=*lista;
        while(p->sgte!=(*lista)){
          p=p->sgte;
```

```
        }
        *lista=(*lista)->sgte;
        p->sgte=*lista;
      }
      cout<<"Se eliminó el dato inicial de la lista"<<endl;
   }
   else{ /** Si la lista está vacía mandar mensaje **/
      cout<<"Lista vacia"<<endl;
   }
}


void eliminarfinal(nodo **lista){
   nodo *p;
   nodo *a;
   if(*lista!=NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
      if(*lista==(*lista)->sgte){ /** Si la lista es una sola, entoces apuntará a NULL **/
        *lista=NULL;
      }
      else{ /** En todo caso, se realizará el proceso de eliminar al final, siempre enlazando el primero con el final **/
        p=*lista;
        while(p->sgte!=(*lista)){
          a=p;
          p=p->sgte;
        }
        a->sgte=*lista;
      }
      cout<<"Se elimino el dato final de la lista"<<endl;
   }
   else{ /** Si la lista está vacía mandar mensaje **/
      cout<<"Lista vacia"<<endl;
```

```
  }
}


void eliminarposicion(nodo **lista, int pos){
  nodo *q;
  nodo *a;
  nodo *b;
  int i=1, cant=1;
  q=*lista;
  a=*lista;
  if(*lista!=NULL){
    while(a->sgte!=(*lista)){ /** Se halla la cantidad de nodos/datos de la lista **/
      a=a->sgte;
      cant++;
    }
    if(pos==1){ /** Si la posicion a eliminar es inicial llamar a eliminarinicio **/
      eliminarinicio(&(*lista));
    }
    else if(pos==cant){ /** Si la posicion es la posicion final de la lista llamar a
eliminarfinal **/
        eliminarfinal(&(*lista));
    }
    else if(pos<cant && pos>1) { /** Hallo un caso general para cualquier posicion
que se encuentre entre la primera posición y ´ltima posición **/
        if(pos<=cant && pos>1){
          while(i!=pos){
            b=q;
            q=q->sgte;
            i++;
        }
        b->sgte=q->sgte;
      }
    }
```

```
    if(pos<=cant && pos>=1){ /** Mandar mensaje si se cumplió que la posición está dentro de los rangos de la lista **/
      cout<<"Posicion eliminada con exito"<<endl;
    }
    else{ /** En todo caso no se pudó realizar ninguna operación y mandar mensaje **/
      cout<<"Posicion inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }
}


void eliminardato(nodo **lista, int dato){ /** Similar a la funcion eliminarposicion **/
  nodo *p;                    /** Solo ubicaré la posición del dato y el resto lo haré como la función eliminarposicion **/
  nodo *q;
  nodo *a;
  nodo *b;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  p=*lista;
  if(*lista!=NULL){
    while(a->sgte!=(*lista)){
      a=a->sgte;
      cant++;
    }
    while(p->dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el dato **/
      p=p->sgte;
```

```
      pos++;
    }
    if(cant>=pos || pos==1){
      if(pos==1){          /** Este procedimiento a seguir es igual a la de la funcion
eliminarposicion **/
        eliminarinicio(&(*lista));
      }
      else {
        if(pos==cant){
          eliminarfinal(&(*lista));
        }
        else{
          if(pos<cant && pos>1){
            while(i!=pos){
              b=q;
              q=q->sgte;
              i++;
          }
            b->sgte=q->sgte;
          }
        }
      }
    }
    if(pos<=cant && pos>=1){ /** Mensajes de aviso si el dato existe o no **/
      cout<<dato<<" eliminado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }
```

```
}


void buscardato(nodo *lista, int dato){
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if(lista!=NULL){
    q=lista;
    while(q->sgte!=lista){ /** Hallo la cantidad de nodos de la lista **/
      q=q->sgte;
      a++;
    }
    p=lista;
    while(p->dato!=dato && p->sgte!=lista && a>=i){ /** Realizo operaciones para buscar el dato **/
      p=p->sgte;
      i++;
    }
    if(i==a){
      if(p->dato!=dato){ /** Si el nodo p llega al último dato **/
        i=a+1;
      }
    }
    if(a>=i){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      cout<<dato<<" se encuentra en la posicion: "<<i;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
  }
  else{
    cout<<"La lista esta vacia"<<endl;
```

```
  }
}


void editardato(nodo **lista, int dato, int dato2){
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if((*lista)!=NULL){
    q=*lista;
    while(q->sgte!=(*lista)){ /** Hallo la cantidad de nodos de la lista **/
      q=q->sgte;
      a++;
    }
    p=*lista;
    while(p->dato!=dato && p->sgte!=(*lista) && a>=i){ /** Realizo operaciones para buscar el dato **/
      p=p->sgte;
      i++;
    }
    if(i==a){
      if(p->dato!=dato){ /** Si el nodo p llega al último dato **/
        i=a+1;
      }
    }
    if(p->dato==dato){ /** Si el p->dato es igual al dato encontrado entonces se modifica **/
      p->dato=dato2;
    }
    if(a>=i){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      cout<<dato<<" se encuentro en la posicion: "<<i<< "y se edito por "<<dato2;
    }
    else{
```

```
        cout<<dato<<" no se encuentra en la lista"<<endl;
      }
    }
    else{
      cout<<"La lista esta vacia"<<endl;
    }
}


void ordenardato(nodo *lista){ /** Ordenamiento por el método de burbuja asociada a listas **/
    int a=1, k=1, j, t;
    nodo *p;
    nodo *q;
    nodo *b;
    if(lista!=NULL){
    p=lista;
    b=lista;
    while(b->sgte!=lista){ /** Hallamos la cantidad de elementos de la lista por ser circular, si hubiese trabajado con simples solo sería con NULL**/
      b=b->sgte;
      a++;
    }
    while(k!=a+1){ /** En este método trabajamos con el nodo sgte conjuntamente con el total de elementos de la lista **/
      q=p->sgte;
      j=1;
      while(j!=a){
        if((p->dato)<(q->dato)){
          t=q->dato;
          q->dato=p->dato;
          p->dato=t;
        }
        q=q->sgte;
```

```
        j++;
      }
      p=p->sgte;
      k++;
    }
    cout<<"Elementos ordenados ascendentemente"<<endl;
    }
    else{
      cout<<"Lista vacia"<<endl;
    }
}


void cantidad(nodo *lista){
    int cant=1;
    nodo *p;
    p=lista;
    if(lista!=NULL){
      while(p->sgte!=lista){ /** Cuenta los elementos yendo a través de cada dato **/
        p=p->sgte;
        cant++;
      }
      cout<<"La lista posee: "<<cant<<" elementos"<<endl;
    }
    else{
      cout<<"La lista esta vacia"<<endl;
    }
}


void guardar(nodo **lista){ /** Guardar el archivo de la lista circular en forma de dato **/
    FILE *H;
```

```
  nodo *p;
  nodo *q;
  int dato,a=1, j=0;
  p=*lista;
  q=*lista;
  if(*lista!=NULL){
    H=fopen("ListaSimCir2.dat","w+");
    while(q->sgte!=(*lista)){
      q=q->sgte;
      a++;
    }
    while(a!=j){
      dato=p->dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=p->sgte;
      j++;
    }
    fclose(H);
    cout<<"Archivo guardado"<<endl;
  }
  else{
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
}

void recuperar(nodo **lista){ /** Se recuperan los datos y se insertar al final en una
lista circular **/
  FILE *H;
  int dato;
  H=fopen("ListaSimCir2.dat","r+");
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
```

```
  fread(&dato,sizeof(dato),1,H);
  while(!feof(H)){
    insertarfinal(&(*lista), dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

ANEXO 4.3.4

4.3.4 Vista panorámica del menú de las listas simples circulares estáticas:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define max 100 /** Define la cantidad máxima de listas estáticas con la que voy a trabajar **/

using namespace std;

struct nodo{ /** Declaro una estructura en forma de lista simple simple estática **/
  int dato;
  int sgte;
};

void crear(nodo lista[max], int *cab, int *cab1); /** Funciones cabeceras **/
void nuevonodo(int *p, nodo lista[max], int *cab1);
void mostrarlista(nodo lista[max], int cab);
void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato);
void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato);
void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos);
void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2);
void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2);
void eliminarinicio(nodo lista[max], int *cab);
void eliminarfinal(nodo lista[max], int *cab);
void eliminarposicion(nodo lista[max], int *cab, int pos);
void eliminardato(nodo lista[max], int *cab, int dato);
void buscardato(nodo lista[max], int cab, int dato);
void editardato(nodo lista[max], int cab, int dato, int dato2);
void ordenardato(nodo lista[max], int cab);
void cantidad(nodo lista[max], int cab);
void guardar(nodo lista[max], int cab);
void recuperar(nodo lista[max], int *cab, int *cab1);
```

```
int main(){
  nodo lista[max]; /** Declaro una variable de la lista estática simple **/
  int cab, cab1, opc, dato, dato2, pos; /** La cabecera de la lista será una variable cab
**/
  crear(lista,&cab,&cab1); /** cab1 ayudará a la creación de nuevos nodos **/
  do{
    system("color 1B");
    cout<<"\n\n\n";
    cout<<("*****MENU*****")<<endl; /** Por aquí se visualiza el MENU **/
    cout<<"1.  Mostrar lista"<<endl;
    cout<<"2.  Insertar al inicio"<<endl;
    cout<<"3.  Insertar al final"<<endl;
    cout<<"4.  Insertar en una posicion determinada"<<endl;
    cout<<"5.  Insertar antes de un dato"<<endl;
    cout<<"6.  Insertar despues de un dato"<<endl;
    cout<<"7.  Eliminar al inicio"<<endl;
    cout<<"8.  Eliminar al final"<<endl;
    cout<<"9.  Eliminar en una posicion determinada"<<endl;
    cout<<"10. Eliminar un dato"<<endl;
    cout<<"11. Buscar dato"<<endl;
    cout<<"12. Editar dato"<<endl;
    cout<<"13. Ordenar lista"<<endl;
    cout<<"14. Numero de elementos de la lista"<<endl;
    cout<<"15. Guardar lista"<<endl;
    cout<<"16. Recuperar lista"<<endl;
    cout<<"17. Salir"<<endl;
    cout<<"Ingrese opcion: ";
    cin>>opc;
    cout<<"\n\n\n";
    switch(opc){
      case 1:  cout<<"LISTADO: "<<endl;
               mostrarlista(lista,cab);
```

```
        break;
case 2:  cout<<"Valor a insertar al inicio: ";
        cin>>dato;
        insertarinicio(lista,&cab,&cab1,dato);
        break;
case 3:  cout<<"Valor a insertar al final: ";
        cin>>dato;
        insertarfinal(lista,&cab,&cab1,dato);
        break;
case 4:  cout<<"Valor a insertar: ";
        cin>>dato;
        cout<<"En que posicion?: ";
        cin>>pos;
        insertarposicion(lista,&cab,&cab1,dato,pos);
        break;
case 5:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Antes del dato: ";
        cin>>dato2;
        insertarantesdato(lista,&cab,&cab1,dato,dato2);
        break;
case 6:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Despues del dato: ";
        cin>>dato2;
        insertardespuesdato(lista,&cab,&cab1,dato,dato2);
        break;
case 7:  eliminarinicio(lista,&cab);
        break;
case 8:  eliminarfinal(lista,&cab);
        break;
case 9:  cout<<"Posicion a eliminar: ";
        cin>>pos;
        eliminarposicion(lista,&cab,pos);
```

```
            break;
        case 10:  cout<<"Dato a eliminar: ";
            cin>>dato;
            eliminardato(lista,&cab,dato);
            break;
        case 11: cout<<"Dato a buscar: ";
            cin>>dato;
            buscardato(lista,cab,dato);
            break;
        case 12: cout<<"Dato a buscar: ";
            cin>>dato;
            cout<<"Editar por: ";
            cin>>dato2;
            editardato(lista, cab, dato, dato2);
            break;
        case 13: ordenardato(lista,cab);
            break;
        case 14: cantidad(lista, cab);
            break;
        case 15: guardar(lista,cab);
            break;
        case 16: recuperar(lista,&cab,&cab1);
            break;
      }
   }
   while(opc!=17);
   system("pause");
   return(0);
}
```

void crear(nodo lista[max], int *cab, int *cab1){ /** Crea todas las estructuras posibles con el límite máximo declarado simulando la creación de espacio de nodos **/

```
  int i;
  *cab=-1;
  *cab1 =0;
  lista[0].sgte =1;
  for(i=1;i<max-1;i++)
  {
    lista[i].dato=0;
    lista[i].sgte = i+1;
  }
  lista[max-1].sgte =0;
}

void nuevonodo(int *p, nodo lista[max], int *cab1) /** Crea un nuevo nodo a partir de la posicion de cab1 y la función crear **/
{
  *p = *cab1;
  /** if(*cab1 == 0)
  {
    printf(" No hay suficiente memoria\n\n");
    exit(1);
  } **/
  *cab1 = lista[*cab1].sgte;
}


void mostrarlista(nodo lista[max], int cab){ /** Muestra cada nodo visualizado a partir de las estructuras creadas hasta el ultimo momento **/
  int q;
  q=cab;
  if(q!=-1){
    cout<<lista[q].dato<<endl;
    q=lista[q].sgte;
    while(q!=cab){
      cout<<lista[q].dato<<endl;
```

```
      q=lista[q].sgte;
    }
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato){ /** Inserta al inicio de la lista **/
  int p,q;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo sgte apunta a si mismo **/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=q;
    *cab=q;
  }
  else{  /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el ultimo nodo-sgte apunten a si mismo **/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    p=*cab;
    while(lista[p].sgte!=*cab){
      p=lista[p].sgte;
    }
    lista[p].sgte=q;
    lista[q].sgte=*cab;
    *cab=q;
  }
  cout<<dato<<" insertado al inicio de la lista "<<endl;
}
```

```
void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato){ /** Inserta al final de la lista **/
  int q,p;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo sgte apunta a si mismo **/
     nuevonodo(&q,lista,&(*cab1));
     lista[q].dato=dato;
     lista[q].sgte=q;
     *cab=q;
  }
  else{ /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el ultimo nodo-sgte apunte a si mismo **/
     nuevonodo(&q,lista,&(*cab1));
     p=*cab;
     lista[q].dato=dato;
     while(lista[p].sgte!=(*cab)){
       p=lista[p].sgte;
     }
     lista[p].sgte=q;
     lista[q].sgte=*cab;
  }
  cout<<dato<<" insertado al final de la lista "<<endl;
}


void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos){ /** Inserta en una posición dada y y reorganiza los elementos de la lista **/
  int i=1, cant=1 , p, q ,a ,b; /**Nodos auxiliares **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  if(pos==1){ /** Si la lista está vacia o se inserta a la 1° posición se llama a la función insertarinicio **/
```

```
    insertarinicio(lista, &(*cab), &(*cab1), dato);
  }
  else {
    while(lista[a].sgte!=*cab){ /** Si la lista posee más de un elemento se halla la cantidad de nodos en la lista **/
      a=lista[a].sgte;
      cant++;
    }
    if(pos==cant+1){ /** Si se quiere insertar en la posición sgte. de toda la lista se llama a la función insertarfinal **/
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else if(pos<cant+1 && pos>1){
      lista[p].dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          i++;
        }
      lista[p].sgte=lista[b].sgte;
      lista[b].sgte=p;
    }
  }
  if(pos<=cant+1 && pos>=1){ /** Si la posición está entre los rangos normales de la lista se procede a enviar un mensaje que se insertó **/
    cout<<"Posicion insertada con exito"<<endl;
  }
  else{ /** En todo caso no realiza ninguna operación por estar fuera de los rangos de la lista **/
    cout<<"Posicion inexistente de la lista"<<endl;
  }
}
```

```
void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b ,c, d; /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=*cab){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[c].dato!=dato2 && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=lista[c].sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==1){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarinicio(lista, &(*cab), &(*cab1), dato);
    }
    else{
       if(pos<cant+1 && pos>1){
        lista[p].dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          i++;
        }
        lista[p].sgte=lista[b].sgte;
```

```
          lista[b].sgte=p;
         }
      }

     if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
        cout<<dato<<" insertado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
   }
   else{
     cout<<"Lista vacia!"<<endl;
   }

}


void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
   int i=1, cant=1, pos=1, p, q, a, b ,c, d; /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
   nuevonodo(&p,lista,&(*cab1));
   q=*cab;
   a=*cab;
   c=*cab;
   lista[p].dato=dato;
   if(*cab!=-1){
     while(lista[a].sgte!=*cab){
        a=lista[a].sgte;
        cant++;
      }
```

```
    while(lista[c].dato!=dato2 && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=lista[c].sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){   /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else{
      if(pos<=cant && pos>=1){
        d=lista[q].sgte;
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          d=lista[q].sgte;
          i++;
      }
        lista[q].sgte=p;
        lista[p].sgte=d;
      }
    }

    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }
```

```
}


void eliminarinicio(nodo lista[max], int *cab){ /** Elimina al inicio de la lista **/
  int p;
  if(*cab!=-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if(*cab==lista[*cab].sgte){ /** Si la lista es una sola, entoces apuntará a NULL **/
      *cab=-1;
    }
    else{ /** En todo caso, se realizará el proceso de eliminar al inicio, siempre enlazando el primero con el final **/
      p=*cab;
      while(lista[p].sgte!=(*cab)){
        p=lista[p].sgte;
      }
      *cab=lista[*cab].sgte;
      lista[p].sgte=*cab;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void eliminarfinal(nodo lista[max], int *cab){ /** Elimina al final de la lista **/
  int p,a;
  p=*cab;
  a=*cab;
```

```
  if(*cab!=-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if(*cab==lista[*cab].sgte){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
      *cab=-1;
    }
    else{ /** En todo caso, se realizará el proceso de eliminar al final, siempre enlazando el primero con el final **/
      while(lista[p].sgte!=*cab){
        a=p;
        p=lista[p].sgte;
      }
      lista[a].sgte=*cab;
    }
    cout<<"Se elimino el dato final de la lista"<<endl;
  }
  else{
    cout<<"No existe lista"<<endl;
  }
}


void eliminarposicion(nodo lista[max], int *cab, int pos){ /** Elimina una posición dada y reorganiza la lista **/
  int i=1, cant=1, p, q, a, b,c;
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=*cab){ /** Se halla la cantidad de nodos/datos de la lista **/
      a=lista[a].sgte;
      cant++;
    }
    if(pos==1){ /** Si la posicion a eliminar es inicial llamar a eliminarinicio **/
```

```
        eliminarinicio(lista,&(*cab));
      }
      else if(pos==cant){ /** Si la posicion es la posicion final de la lista llamar a eliminarfinal **/
          eliminarfinal(lista,&(*cab));
      }
      else if(pos<cant && pos>1) { /** Hallo un caso general para cualquier posicion que se encuentre entre la primera posición y ´ltima posición **/
          while(i!=pos){
            b=q;
            q=lista[q].sgte;
            c=lista[q].sgte;
            i++;
        }
        lista[b].sgte=c;
      }
      if(pos<=cant && pos>=1){ /** Mandar mensaje si se cumplió que la posición está dentro de los rangos de la lista **/
        cout<<"Posicion eliminada con exito"<<endl;
      }
      else{ /** En todo caso no se pudó realizar ninguna operación y mandar mensaje **/
        cout<<"Posicion inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }
}


void eliminardato(nodo lista[max], int *cab, int dato){ /** Similar a la funcion eliminarposicion **/
```

```
  int i=1, cant=1, pos=1, p, q, a, b, c; /** Solo ubicaré la posición del dato y el resto lo haré como la función eliminarposicion **/
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=*cab){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      p=lista[p].sgte;
      pos++;
    }
    if(cant>=pos || pos==1){
      if(pos==1){           /** Este procedimiento a seguir es igual a la de la funcion eliminarposicion **/
        eliminarinicio(lista,&(*cab));
      }
      else {
        if(pos==cant){
          eliminarfinal(lista,&(*cab));
        }
        else{
          if(pos<cant && pos>1){
            while(i!=pos){
              b=q;
              q=lista[q].sgte;
              c=lista[q].sgte;
              i++;
            }
            lista[b].sgte=c;
          }
```

```
        }
      }
    }
    if(pos<=cant && pos>=1){ /** Mensajes de aviso si el dato existe o no **/
      cout<<dato<<" eliminado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}


void buscardato(nodo lista[max], int cab, int dato){ /** Busca una dato y avisa si se encuentra(con su posición) **/
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=cab){ /** Hallo la cantidad de nodos de la lista **/
      cant++;
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el dato **/
      pos++;
      p=lista[p].sgte;
    }
```

```
  if(cant>=pos && pos>=1){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
    cout<<dato<<" se encontro en la posicion "<<pos<<endl;
  }
  else{
    cout<<"Dato inexistente en la lista"<<endl;
  }
  }
}


void editardato(nodo lista[max], int cab, int dato, int dato2){ /** Similar a buscardato, reemplaza el dato pedido por el dato anterior de la lista **/
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=cab){ /** Hallo la cantidad de nodos de la lista **/
      cant++;
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el dato **/
      pos++;
      p=lista[p].sgte;
    }
    if(cant>=pos){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      lista[p].dato=dato2; /** Aquí edito el dato último seleccionado de "p" **/
      cout<<dato<<" se encuentro en la posicion: "<<pos<< " y se edito por "<<dato2;
    }
    else{
```

```
        cout<<dato<<" no se encuentra en la lista"<<endl;
      }

   }
}


void cantidad(nodo lista[max], int cab){ /** Determina la cantidad de elementos de la lista **/
   int p, cant=1;
   p=cab;
   if(cab!=-1){
      while(lista[p].sgte!=cab){ /** Cuenta los elementos yendo a través de cada dato **/
         p=lista[p].sgte;
         cant++;
      }
      cout<<"La lista posee: "<<cant<<" elementos"<<endl;
   }
   else{
      cout<<"La lista esta vacia"<<endl;
   }
}


void ordenardato(nodo lista[max], int cab){ /** Ordenamiento por el método de burbuja asociada a listas **/
   int p,b,q,a=1, k=1, j, t;
   if(cab!=-1){
      p=cab;
      b=cab;
      q=cab;
      while(lista[b].sgte!=cab){ /** Hallamos la cantidad de elementos de la lista por ser circular, si hubiese trabajado con simples solo sería con NULL**/
```

```
        b=lista[b].sgte;
        a++;
      }
      while(k!=a+1){ /** En este método trabajamos con el nodo sgte conjuntamente con el total de elementos de la lista **/
        q=lista[p].sgte;
        j=1;
        while(j!=a){
          if((lista[p].dato)<(lista[q].dato)){
            t=lista[q].dato;
            lista[q].dato=lista[p].dato;
            lista[p].dato=t;
          }
          q=lista[q].sgte;
          j++;
        }
        p=lista[p].sgte;
        k++;
      }
      cout<<"Elementos ordenados ascendentemente"<<endl;
    }
    else{
      cout<<"Lista vacia"<<endl;
    }
}


void guardar(nodo lista[max], int cab){ /** Guardar el archivo de la lista simple circular en forma de dato **/
    FILE *H;
    int p,q,dato,a=1, j=0;
    p=cab;
    q=cab;
    if(cab!=-1){
```

```
    H=fopen("ListaSimCir.dat","w+");
    while(lista[q].sgte!=(cab)){
      q=lista[q].sgte;
      a++;
    }
    while(a!=j){
      dato=lista[p].dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=lista[p].sgte;
      j++;
    }
    fclose(H);
    cout<<"Archivo guardado"<<endl;
  }
  else{
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
}


void recuperar(nodo lista[max], int *cab, int *cab1){ /** Se recuperan los datos y se
insertar al final en una lista simple circular estática **/
  int p, dato;
  FILE *H;
  p=*cab;
  H=fopen("ListaSimCir.dat","r+");
  fread(&dato,sizeof(dato),1,H);
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
  while(!feof(H)){
    insertarfinal(lista,&(*cab),&(*cab1),dato);
    fread(&dato,sizeof(dato),1,H);
```

```
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

ANEXO 4.3.5

4.3.5 Vista panorámica del menú de las listas simples dobles circulares dinámicas:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

using namespace std;

struct nodo{ /** Declaro una estructura en forma de lista doble circular dinámica **/
  nodo *ant;
  int dato;
  nodo *sgte;
  };

void crearlista(nodo **lista); /** Funciones cabeceras **/
void mostrarlista(nodo *lista);
void insertarinicio(nodo **lista, int dato);
void insertarfinal(nodo **lista, int dato);
void insertarposicion(nodo **lista, int dato, int pos);
void insertarantesdato(nodo **lista, int dato, int dato2);
void insertardespuesdato(nodo **lista, int dato, int dato2);
void eliminarinicio(nodo **lista);
void eliminarfinal(nodo **lista);
void eliminarposicion(nodo **lista, int pos);
void eliminardato(nodo **lista, int dato);
void buscardato(nodo *lista, int dato);
void editardato(nodo **lista, int dato, int dato2);
void ordenardato(nodo *lista);
void cantidad(nodo *lista);
void guardar(nodo **lista);
void recuperar(nodo **lista);
```

```
int main(){
   nodo *lista;
   crearlista(&lista);
   int opc, dato, dato2, pos;
   do{
      system("color 1B");
      //system("cls");
      cout<<"\n\n\n";
      cout<<("*****MENU*****")<<endl; /** Por aquí se visualiza el MENU **/
      cout<<"1.  Mostrar lista"<<endl;
      cout<<"2.  Insertar al inicio"<<endl;
      cout<<"3.  Insertar al final"<<endl;
      cout<<"4.  Insertar en una posicion determinada"<<endl;
      cout<<"5.  Insertar antes de un dato"<<endl;
      cout<<"6.  Insertar despues de un dato"<<endl;
      cout<<"7.  Eliminar al inicio"<<endl;
      cout<<"8.  Eliminar al final"<<endl;
      cout<<"9.  Eliminar en una posicion determinada"<<endl;
      cout<<"10. Eliminar un dato"<<endl;
      cout<<"11. Buscar dato"<<endl;
      cout<<"12. Editar dato"<<endl;
      cout<<"13. Ordenar lista"<<endl;
      cout<<"14. Cantidad de datos"<<endl;
      cout<<"15. Guardar lista"<<endl;
      cout<<"16. Recuperar lista"<<endl;
      cout<<"17. Salir"<<endl;
      cout<<"Ingrese opcion: ";
      cin>>opc;
      cout<<"\n\n\n";
      switch(opc){
        case 1:  cout<<"LISTADO: "<<endl;
                 mostrarlista(lista);
                 break;
        case 2:  cout<<"Valor a insertar al inicio: ";
```

```
        cin>>dato;
        insertarinicio(&lista, dato);
        break;
case 3:  cout<<"Valor a insertar al final: ";
        cin>>dato;
        insertarfinal(&lista, dato);
        break;
case 4:  cout<<"Valor a insertar: ";
        cin>>dato;
        cout<<"En que posicion?: ";
        cin>>pos;
        insertarposicion(&lista, dato, pos);
        break;
case 5:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Antes del dato: ";
        cin>>dato2;
        insertarantesdato(&lista, dato, dato2);
        break;
case 6:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Despues del dato: ";
        cin>>dato2;
        insertardespuesdato(&lista, dato, dato2);
        break;
case 7:  eliminarinicio(&lista);
        break;
case 8:  eliminarfinal(&lista);
        break;
case 9:  cout<<"Posicion a eliminar: ";
        cin>>pos;
        eliminarposicion(&lista, pos);
        break;
case 10: cout<<"Dato a eliminar: ";
```

```
                cin>>dato;
                eliminardato(&lista, dato);
                break;
        case 11: cout<<"Dato a buscar: ";
                cin>>dato;
                buscardato(lista, dato);
                break;
        case 12: cout<<"Dato a buscar: ";
                cin>>dato;
                cout<<"Editar por: ";
                cin>>dato2;
                editardato(&lista, dato, dato2);
                break;
        case 13: ordenardato(lista);
                break;
        case 14: cantidad(lista);
                break;
        case 15: guardar(&lista);
                break;
        case 16: recuperar(&lista);
                break;
      }
   }
   while(opc!=17);
   system("pause");
   return(0);
}



void crearlista(nodo **lista){
   *lista=NULL;
}
```

```
void mostrarlista(nodo *lista){ /** Muestra cada nodo visualizado a partir de las estructuras creadas hasta el ultimo momento **/
  nodo *p;
  if(lista!=NULL){ /** Pregunta si la lista está vacía */
    p=lista; /** El nodo p apunta a la lista */
    cout<<p->dato<<endl; /** Leo el primer dato */
    p=p->sgte;  /** Apunto al siguiente nodo */
    while(p!=lista){ /** Leo los demás datos comenzando a partir del 2° dato hasta que llegue a la lista */
      cout<<p->dato<<endl;
      p=p->sgte;
    }
    cout<<endl;
  }
  else{ /** Si la lista está vacía retorna mensaje de aviso */
    cout<<"No existe lista"<<endl;
  }
}


void insertarinicio(nodo **lista, int dato){ /** Se inserta dato al inicio de la lista **/
    nodo *q; /** Nodos auxiliares **/
    nodo *p;
    q=new struct nodo;
    q->dato=dato;
    p=*lista;
    if(*lista==NULL){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a si mismo**/
      q->sgte=q;
      q->ant=q;
      *lista=q;
    }
```

```
    else{   /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el primer nodo-ant del primer nodo apunte al ultimo nodo-sgte del ult. nodo y viceversa **/
        while(p->sgte!=*lista){
          p=p->sgte;
        }
        (*lista)->ant=q;
        q->sgte=*lista;
        q->ant=p;
        p->sgte=q;
        *lista=q;
    }
    cout<<dato<<" insertado al inicio de la lista "<<endl;
}


void insertarfinal(nodo **lista, int dato){
    nodo *p;
    nodo *q;
    q=new struct nodo;
    if(*lista==NULL){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a si mismo**/
        q->dato=dato;
        q->sgte=q;
        q->ant=q;
        *lista=q;
    }
    else{   /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el primer nodo-ant del primer nodo apunte al ultimo nodo-sgte del ult. nodo y viceversa **/
        p=*lista;
        q->dato=dato;
        while(p->sgte!=*lista){
          p=p->sgte;
```

```
        }
        q->ant=p;
        p->sgte=q;
        q->sgte=*lista;
        (*lista)->ant=q;
      }
      cout<<dato<<" insertado al inicio de la lista "<<endl;
}


void insertarposicion(nodo **lista, int dato, int pos){
   nodo *p; /**Nodos auxiliares **/
   nodo *q;
   nodo *a;
   nodo *b;
   p=new struct nodo;
   int i=1, cant=1, aux;
   q=*lista;
   a=*lista;
   if(pos==1){ /** Si la lista está vacia o se inserta a la 1° posición se llama a la función insertarinicio **/
      insertarinicio(&(*lista),dato);
   }
   else {
      while(a->sgte!=*lista){ /** Si la lista posee más de un elemento se halla la cantidad de nodos en la lista **/
        a=a->sgte;
        cant++;
      }
      if(pos==cant+1){ /** Si se quiere insertar en la posición sgte. de toda la lista se llama a la función insertarfinal **/
        insertarfinal(&(*lista),dato);
      }
      else if(pos<cant+1 && pos>1){
```

```
        p->dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
        if(pos<=cant && pos>1){
          while(i!=pos){
            b=q;
            q=q->sgte;
            i++;
          }
          p->sgte=b->sgte;
          p->ant=b;
          b->sgte=p;
          q->ant=p;
        }
      }
    }
    if(pos<=cant+1 && pos>=1){ /** Si la posición está entre los rangos normales de la lista se procede a enviar un mensaje que se insertó **/
      cout<<"Posicion insertada con exito"<<endl;
    }
    else{ /** En todo caso no realiza ninguna operación por estar fuera de los rangos de la lista **/
      cout<<"Posicion inexistente de la lista"<<endl;
    }
}


void insertarantesdato(nodo **lista, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
    nodo *p;                    /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
    nodo *q;
    nodo *a;
    nodo *b;
    nodo *c;
```

```
  nodo *d;
  p=new struct nodo;
  p->dato=dato;
  int i=1, cant=1, pos=1;
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=*lista){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=c->sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){
      if(c->dato!=dato2){
        pos=cant+1;
      }
    }
    if(pos==1){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarinicio(&(*lista),dato);
    }
    else{
      if(pos<=cant && pos>=1){
        while(i!=pos){
          b=q;
          q=q->sgte;
          i++;
        }
        p->sgte=b->sgte;
```

```
            p->ant=b;
            b->sgte=p;
            q->ant=p;
          }
        }

      if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
          cout<<dato<<" insertado con exito"<<endl;
        }
        else{
          cout<<"Dato inexistente de la lista"<<endl;
        }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }

}



void insertardespuesdato(nodo **lista, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
    nodo *p;                    /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
    nodo *q;
    nodo *a;
    nodo *b;
    nodo *c;
    nodo *d;
    p=new struct nodo;
    p->dato=dato;
    int i=1, cant=1, pos=1;
```

```
  q=*lista;
  a=*lista;
  c=*lista;
  if(*lista!=NULL){
    while(a->sgte!=*lista){
      a=a->sgte;
      cant++;
    }
    while(c->dato!=dato2 && c->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=c->sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){
      if(c->dato!=dato2){
        pos=cant+1;
      }
    }
    if(pos==cant){   /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarfinal(&(*lista),dato);
    }
    else{
      if(pos<=cant && pos>=1){
        d=q->sgte;
        while(i!=pos){
          b=q;
          q=q->sgte;
          d=q->sgte;
          i++;
      }
        q->sgte=p;
        p->ant=q;
        p->sgte=d;
```

```
        d->ant=p;
      }
    }

    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}
```

```
void eliminarinicio(nodo **lista){
  nodo *p;
  nodo *q;
  if(*lista!=NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if((*lista)->sgte==*lista){ /** Si la lista es una sola, entoces apuntará a NULL **/
      *lista=NULL;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo, de tal manera que el primer y último nodo se enlazan **/
      q=(*lista)->ant;
```

```
      p=(*lista)->sgte;
      p->ant=q;
      q->sgte=p;
      *lista=p;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void eliminarfinal(nodo **lista){
  nodo *p;
  nodo *a;
  p=*lista;
  a=*lista;
  if(*lista!=NULL){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if((*lista)->sgte==*lista){ /** Si la lista es una sola, entoces apuntará a NULL **/
      *lista=NULL;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo, de tal manera que el primer y último nodo se enlazan **/
      while(p->sgte!=*lista){
        a=p;
        p=p->sgte;
      }
      a->sgte=*lista;
      (*lista)->ant=a;
    }
    cout<<"Se elimino el dato final de la lista"<<endl;
```

```
  }
  else{
    cout<<"No existe lista"<<endl;
  }
}


void eliminarposicion(nodo **lista, int pos){
  nodo *q;
  nodo *a;
  nodo *b;
  nodo *c;
  int i=1, cant=1;
  q=*lista;
  a=*lista;
  if(*lista!=NULL){
    while(a->sgte!=*lista){ /** Se halla la cantidad de nodos/datos de la lista **/
      a=a->sgte;
      cant++;
    }
    if(pos==1){ /** Si la posicion a eliminar es inicial llamar a eliminarinicio **/
      eliminarinicio(&(*lista));
    }
    else if(pos==cant){ /** Si la posicion es la posicion final de la lista llamar a
eliminarfinal **/
        eliminarfinal(&(*lista));
    }
    else if(pos<cant && pos>1) { /** Hallo un caso general para cualquier posicion
que se encuentre entre la primera posición y última posición **/
        if(pos<=cant && pos>1){
          while(i!=pos){
            b=q;
            q=q->sgte;
            c=q->sgte;
```

```
            i++;
          }
          b->sgte=c;
          c->ant=b;
        }
      }
      if(pos<=cant && pos>=1){ /** Mandar mensaje si se cumplió que la posición está dentro de los rangos de la lista **/
        cout<<"Posicion eliminada con exito"<<endl;
      }
      else{ /** En todo caso no se pudó realizar ninguna operación y mandar mensaje **/
        cout<<"Posicion inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }
}


void eliminardato(nodo **lista, int dato){ /** Similar a la funcion eliminarposicion **/
    nodo *p;                    /** Solo ubicaré la posición del dato y el resto lo haré como la función eliminarposicion **/
    nodo *q;
    nodo *a;
    nodo *b;
    nodo *c;
    int i=1, cant=1, pos=1;
    q=*lista;
    a=*lista;
    p=*lista;
    if(*lista!=NULL){
```

```
    while(a->sgte!=*lista){
      a=a->sgte;
      cant++;
    }
    while(p->dato!=dato && p->sgte!=NULL && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      p=p->sgte;
      pos++;
    }
    if(pos==cant){
      if(p->dato!=dato){
        pos=cant+1;
      }
    }
    if(cant>=pos || pos==1){
      if(pos==1){           /** Este procedimiento a seguir es igual a la de la funcion eliminarposicion **/
        eliminarinicio(&(*lista));
      }
      else {
        if(pos==cant){
          eliminarfinal(&(*lista));
        }
        else{
          if(pos<cant && pos>1){
            while(i!=pos){
              b=q;
              q=q->sgte;
              c=q->sgte;
              i++;
          }
            b->sgte=c;
            c->ant=b;
          }
```

```
        }
      }
    }
    if(pos<=cant && pos>=1){ /** Mensajes de aviso si el dato existe o no **/
      cout<<dato<<" eliminado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
  }

}



void buscardato(nodo *lista, int dato){
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if(lista!=NULL){
    q=lista;
    while(q->sgte!=lista){ /** Hallo la cantidad de nodos de la lista **/
      q=q->sgte;
      a++;
    }
    p=lista;
    while(p->dato!=dato && p->sgte!=NULL && a>=i){ /** Realizo operaciones
para buscar el dato **/
      p=p->sgte;
      i++;
    }
```

```
    if(i==a){ /** Si el nodo p llega al último dato **/
      if(p->dato!=dato){
        i=a+1;
      }
    }
    if(a>=i){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      cout<<dato<<" se encuentra en la posicion: "<<i;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}


void editardato(nodo **lista, int dato, int dato2){
  nodo *p;
  nodo *q;
  int i=1, a=1;
  if((*lista)!=NULL){
    q=*lista;
    while(q->sgte!=*lista){ /** Hallo la cantidad de nodos de la lista **/
      q=q->sgte;
      a++;
    }
    p=*lista;
    while(p->dato!=dato && p->sgte!=NULL && a>=i){ /** Realizo operaciones para buscar el dato **/
      p=p->sgte;
      i++;
```

```
    }
    if(i==a){ /** Si el nodo p llega al último dato **/
      if(p->dato!=dato){
        i=a+1;
      }
    }
    if(p->dato==dato){ /** Si el p->dato es igual al dato encontrado entonces se modifica **/
      p->dato=dato2;
    }
    if(a>=i){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
      cout<<dato<<" se encuentro en la posicion: "<<i<< "y se edito por "<<dato2;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}


void ordenardato(nodo *lista){ /** Ordenamiento por el método de burbuja asociada a listas **/
  int a=1, k=1, j, t;
  nodo *p;
  nodo *q;
  nodo *b;
  if(lista!=NULL){
  p=lista;
  b=lista;
```

```
  while(b->sgte!=lista){ /** Hallamos la cantidad de elementos de la lista por ser circular, si hubiese trabajado con simples solo sería con NULL**/
    b=b->sgte;
    a++;
  }
  while(k!=a+1){ /** En este método trabajamos con el nodo sgte conjuntamente con el total de elementos de la lista **/
    q=p->sgte;
    j=1;
    while(j!=a){
      if((p->dato)<(q->dato)){
        t=q->dato;
        q->dato=p->dato;
        p->dato=t;
      }
      q=q->sgte;
      j++;
    }
    p=p->sgte;
    k++;
  }
  cout<<"Elementos ordenados ascendentemente"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void cantidad(nodo *lista){
  int cant=1;
  nodo *p;
  p=lista;
  if(lista!=NULL){
```

```
    while(p->sgte!=lista){ /** Cuenta los elementos yendo a través de cada dato **/
      p=p->sgte;
      cant++;
    }
    cout<<"La lista posee: "<<cant<<" elementos"<<endl;
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}



void guardar(nodo **lista){ /** Guardar el archivo de la lista circular en forma de dato **/
  FILE *H;
  nodo *p;
  nodo *q;
  int dato,a=1, j=0;
  p=*lista;
  q=*lista;
  if(*lista!=NULL){
    H=fopen("ListaDobCir2.dat","w+");
    while(q->sgte!=(*lista)){
      q=q->sgte;
      a++;
    }
    while(a!=j){
      dato=p->dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=p->sgte;
      j++;
    }
    fclose(H);
```

```
    cout<<"Archivo guardado"<<endl;
  }
  else{
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
}

void recuperar(nodo **lista){ /** Se recuperan los datos y se insertar al final en una lista circular **/
  FILE *H;
  int dato;
  H=fopen("ListaDobCir2.dat","r+");
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
  fread(&dato,sizeof(dato),1,H);
  while(!feof(H)){
    insertarfinal(&(*lista), dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

ANEXO 4.3.6

4.3.6 Vista panorámica del menú de las listas simples dobles circulares estáticas:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define max 100 /** Define la cantidad máxima de listas estáticas con la que voy a trabajar **/

using namespace std;

struct nodo{ /** Declaro una estructura en forma de lista doble estática **/
  int ant;
  int dato;
  int sgte;
};

void crear(nodo lista[max], int *cab, int *cab1); /** Funciones cabeceras **/
void nuevonodo(int *p, nodo lista[max], int *cab1);
void mostrarlista(nodo lista[max], int cab);
void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato);
void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato);
void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos);
void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2);
void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2);
void eliminarinicio(nodo lista[max], int *cab);
void eliminarfinal(nodo lista[max], int *cab);
void eliminarposicion(nodo lista[max], int *cab, int pos);
void eliminardato(nodo lista[max], int *cab, int dato);
void buscardato(nodo lista[max], int cab, int dato);
void editardato(nodo lista[max], int cab, int dato, int dato2);
void ordenardato(nodo lista[max], int cab);
void cantidad(nodo lista[max], int cab);
void guardar(nodo lista[max], int cab);
```

```
void recuperar(nodo lista[max], int *cab, int *cab1);


int main(){
  nodo lista[max]; /** Declaro una variable de la lista estática simple **/
  int cab, cab1, opc, dato, dato2, pos; /** La cabecera de la lista será una variable cab
**/
  crear(lista,&cab,&cab1); /** cab1 ayudará a la creación de nuevos nodos **/
  do{
    system("color 1B");
    cout<<"\n\n\n";
    cout<<("*****MENU*****")<<endl; /** Por aquí se visualiza el MENU **/
    cout<<"1.  Mostrar lista"<<endl;
    cout<<"2.  Insertar al inicio"<<endl;
    cout<<"3.  Insertar al final"<<endl;
    cout<<"4.  Insertar en una posicion determinada"<<endl;
    cout<<"5.  Insertar antes de un dato"<<endl;
    cout<<"6.  Insertar despues de un dato"<<endl;
    cout<<"7.  Eliminar al inicio"<<endl;
    cout<<"8.  Eliminar al final"<<endl;
    cout<<"9.  Eliminar en una posicion determinada"<<endl;
    cout<<"10. Eliminar un dato"<<endl;
    cout<<"11. Buscar dato"<<endl;
    cout<<"12. Editar dato"<<endl;
    cout<<"13. Ordenar lista"<<endl;
    cout<<"14. Numero de elementos de la lista"<<endl;
    cout<<"15. Guardar lista"<<endl;
    cout<<"16. Recuperar lista"<<endl;
    cout<<"17. Salir"<<endl;
    cout<<"Ingrese opcion: ";
    cin>>opc;
    cout<<"\n\n\n";
    switch(opc){
      case 1:  cout<<"LISTADO: "<<endl;
```

```
        mostrarlista(lista,cab);
        break;
case 2:  cout<<"Valor a insertar al inicio: ";
        cin>>dato;
        insertarinicio(lista,&cab,&cab1,dato);
        break;
case 3:  cout<<"Valor a insertar al final: ";
        cin>>dato;
        insertarfinal(lista,&cab,&cab1,dato);
        break;
case 4:  cout<<"Valor a insertar: ";
        cin>>dato;
        cout<<"En que posicion?: ";
        cin>>pos;
        insertarposicion(lista,&cab,&cab1,dato,pos);
        break;
case 5:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Antes del dato: ";
        cin>>dato2;
        insertarantesdato(lista,&cab,&cab1,dato,dato2);
        break;
case 6:  cout<<"Dato a ingresar: ";
        cin>>dato;
        cout<<"Despues del dato: ";
        cin>>dato2;
        insertardespuesdato(lista,&cab,&cab1,dato,dato2);
        break;
case 7:  eliminarinicio(lista,&cab);
        break;
case 8:  eliminarfinal(lista,&cab);
        break;
case 9:  cout<<"Posicion a eliminar: ";
        cin>>pos;
```

```
                eliminarposicion(lista,&cab,pos);
                break;
        case 10:  cout<<"Dato a eliminar: ";
                cin>>dato;
                eliminardato(lista,&cab,dato);
                break;
        case 11: cout<<"Dato a buscar: ";
                cin>>dato;
                buscardato(lista,cab,dato);
                break;
        case 12: cout<<"Dato a buscar: ";
                cin>>dato;
                cout<<"Editar por: ";
                cin>>dato2;
                editardato(lista, cab, dato, dato2);
                break;
        case 13: ordenardato(lista,cab);
                break;
        case 14: cantidad(lista, cab);
                break;
        case 15: guardar(lista,cab);
                break;
        case 16: recuperar(lista,&cab,&cab1);
                break;
      }
   }
   while(opc!=17);
   system("pause");
   return(0);
}
```

```
void crear(nodo lista[max], int *cab, int *cab1){   /** Crea todas las estructuras posibles con el límite máximo declarado simulando la creación de espacio de nodos **/
  int i;
  *cab=-1;
  *cab1 =0;
  lista[0].ant =max-1;
  lista[0].sgte =1;
  for(i=1;i<max-1;i++)
  {
    lista[i].dato = 0;
    lista[i].ant=i-1;
    lista[i].sgte = i+1;
  }
  lista[max-1].sgte =0;
  lista[max-1].ant =max-2;
}

void nuevonodo(int *p, nodo lista[max], int *cab1) /** Crea un nuevo nodo a partir de la posicion de cab1 y la función crear **/
{
  *p = *cab1;
  /**if(*cab1 == 1)
  {
    printf(" No hay suficiente memoria\n\n");
    exit(1);
  } **/
  *cab1 = lista[*cab1].sgte;
}


void mostrarlista(nodo lista[max], int cab){ /** Muestra cada nodo visualizado a partir de las estructuras creadas hasta el ultimo momento **/
  int q;
```

```
  q=cab;
  if(q!=-1){
    cout<<lista[q].dato<<endl;
    q=lista[q].sgte;
    while(q!=cab){
      cout<<lista[q].dato<<endl;
      q=lista[q].sgte;
    }
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

```
void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato){ /** Inserta al inicio de la lista **/
  int p,q;
  p=*cab;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a si mismo**/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=q;
    lista[q].ant=q;
    *cab=q;
  }
  else{  /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el primer nodo-ant del primer nodo apunte al ultimo nodo-sgte del ult. nodo y viceversa **/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    while(lista[p].sgte!=*cab){
      p=lista[p].sgte;
```

```
    }
    lista[*cab].ant=q;
    lista[q].sgte=*cab;
    lista[q].ant=p;
    lista[p].sgte=q;
    *cab=q;
    }
  cout<<dato<<" insertado al inicio de la lista "<<endl;
}


void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato){ /** Inserta al final de la lista **/
  int q,p;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a si mismo**/
    nuevonodo(&q,lista,&(*cab1));
    lista[q].dato=dato;
    lista[q].sgte=q;
    lista[q].ant=q;
    *cab=q;
  }
  else{ /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el primer nodo-ant del primer nodo apunte al ultimo nodo-sgte del ult. nodo y viceversa **/
    nuevonodo(&q,lista,&(*cab1));
    p=*cab;
    lista[q].dato=dato;
    while(lista[p].sgte!=*cab){
      p=lista[p].sgte;
    }
    lista[q].ant=p;
    lista[p].sgte=q;
    lista[q].sgte=*cab;
```

```
    lista[*cab].ant=q;
  }
  cout<<dato<<" insertado al final de la lista "<<endl;
}


void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato, int pos){ /** Inserta en una posición dada y y reorganiza los elementos de la lista **/
  int i=1, cant=1 , p, q ,a ,b; /**Nodos auxiliares **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  if(pos==1){ /** Si la lista está vacia o se inserta a la 1° posición se llama a la función insertarinicio **/
    insertarinicio(lista, &(*cab), &(*cab1), dato);
  }
  else {
    while(lista[a].sgte!=*cab){ /** Si la lista posee más de un elemento se halla la cantidad de nodos en la lista **/
      a=lista[a].sgte;
      cant++;
    }
    if(pos==cant+1){ /** Si se quiere insertar en la posición sgte. de toda la lista se llama a la función insertarfinal **/
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else if(pos<cant+1 && pos>1){
      lista[p].dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
      if(pos<=cant && pos>1){
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          i++;
```

```
        }
      lista[p].sgte=lista[b].sgte;
      lista[p].ant=b;
      lista[b].sgte=p;
      lista[q].ant=p;
      }
    }
  }
  if(pos<=cant+1 && pos>=1){ /** Si la posición está entre los rangos normales de la lista se procede a enviar un mensaje que se insertó **/
    cout<<"Posicion insertada con exito"<<endl;
  }
  else{ /** En todo caso no realiza ninguna operación por estar fuera de los rangos de la lista **/
    cout<<"Posicion inexistente de la lista"<<endl;
  }
}


void insertarantesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){ /** Similar a la funcion insertarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b ,c, d; /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=*cab){
      a=lista[a].sgte;
      cant++;
    }
```

```
    while(lista[c].dato!=dato2 && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=lista[c].sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==1){  /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarinicio(lista, &(*cab), &(*cab1), dato);
    }
    else{
       if(pos<cant+1 && pos>1){
        lista[p].dato=dato; /** Si se quiere insertar entre la posición 1 y la cantidad máxima de datos de la lista se procede a realizar la sgte. operación **/
        if(pos<=cant && pos>1){
          while(i!=pos){
            b=q;
            q=lista[q].sgte;
            i++;
          }
          lista[p].sgte=lista[b].sgte;
          lista[p].ant=b;
          lista[b].sgte=p;
          lista[q].ant=p;
        }
        }
    }
    if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
      cout<<dato<<" insertado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
```

```
  else{
    cout<<"Lista vacia!"<<endl;
  }

}


void insertardespuesdato(nodo lista[max], int *cab, int *cab1, int dato, int dato2){
/** Similar a la funcion insertarposicion **/
  int i=1, cant=1, pos=1, p, q, a, b ,c, d; /** Solo ubicaré la posición del dato y el resto lo haré como la función insertarposicion **/
  nuevonodo(&p,lista,&(*cab1));
  q=*cab;
  a=*cab;
  c=*cab;
  lista[p].dato=dato;
  if(*cab!=-1){
    while(lista[a].sgte!=*cab){
      a=lista[a].sgte;
      cant++;
    }
    while(lista[c].dato!=dato2 && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
      c=lista[c].sgte;
      pos++;  /** Ubico la posición del dato de la lista **/
    }
    if(pos==cant){   /** Gracias a la busqueda de la posición del dato puedo realizar las operacionas comodamente con el nodo sgte **/
      insertarfinal(lista, &(*cab), &(*cab1), dato);
    }
    else{
      if(pos<=cant && pos>=1){
        d=lista[q].sgte;
        while(i!=pos){
```

```
            b=q;
            q=lista[q].sgte;
            d=lista[q].sgte;
            i++;
          }
          lista[q].sgte=p;
          lista[p].ant=q;
          lista[p].sgte=d;
          lista[d].ant=p;
        }
      }
      if(pos<=cant && pos>=1){ /** Mensajes de acuerdo al dato existente o inexistente **/
        cout<<dato<<" insertado con exito"<<endl;
      }
      else{
        cout<<"Dato inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }

}


void eliminarinicio(nodo lista[max], int *cab){ /** Elimina al inicio de la lista **/
    int p, q;
    if(*cab!=-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
      if(lista[*cab].sgte==*cab){ /** Si la lista es una sola, entoces apuntará a NULL **/
        *cab=-1;
      }
```

```
    else{ /** En todo caso, se realizará el proceso de liberar el nodo, de tal manera que el primer y último nodo se enlazan **/
      q=lista[*cab].ant;
      p=lista[*cab].sgte;
      lista[p].ant=q;
      lista[q].sgte=p;
      *cab=p;
    }
    cout<<"Se elimino el dato inicial de la lista"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void eliminarfinal(nodo lista[max], int *cab){ /** Elimina al final de la lista **/
  int p,a;
  p=*cab;
  a=*cab;
  if(*cab!=-1){ /** Si la lista no está vacía se realizarán las operaciones de la lista **/
    if(lista[*cab].sgte==*cab){ /** Si la lista es una sola, entoces apuntará a NULL **/
      *cab=-1;
    }
    else{ /** En todo caso, se realizará el proceso de liberar el nodo, de tal manera que el primer y último nodo se enlazan **/
      while(lista[p].sgte!=*cab){
        a=p;
        p=lista[p].sgte;
      }
      lista[a].sgte=*cab;
      lista[*cab].ant=a;
```

```
    }
    cout<<"Se elimino el dato final de la lista"<<endl;
  }
  else{
    cout<<"No existe lista"<<endl;
  }
}
```

void eliminarposicion(nodo lista[max], int *cab, int pos){ /** Elimina una posición dada y reorganiza la lista **/

```
  int i=1, cant=1, p, q, a, b,c;
  q=*cab;
  a=*cab;
  p=*cab;
  if(*cab!=-1){
    while(lista[a].sgte!=*cab){ /** Se halla la cantidad de nodos/datos de la lista **/
      a=lista[a].sgte;
      cant++;
    }
    if(pos==1){ /** Si la posicion a eliminar es inicial llamar a eliminarinicio **/
      eliminarinicio(lista,&(*cab));
    }
    else if(pos==cant){ /** Si la posicion es la posicion final de la lista llamar a eliminarfinal **/
        eliminarfinal(lista,&(*cab));
    }
    else if(pos<cant && pos>1) { /** Hallo un caso general para cualquier posicion que se encuentre entre la primera posición y ´ltima posición **/
        while(i!=pos){
          b=q;
          q=lista[q].sgte;
          c=lista[q].sgte;
          i++;
```

```
        }
        lista[b].sgte=c;
        lista[c].ant=b;
      }
      if(pos<=cant && pos>=1){ /** Mandar mensaje si se cumplió que la posición está dentro de los rangos de la lista **/
        cout<<"Posicion eliminada con exito"<<endl;
      }
      else{ /** En todo caso no se pudó realizar ninguna operación y mandar mensaje **/
        cout<<"Posicion inexistente de la lista"<<endl;
      }
    }
    else{
      cout<<"Lista vacia!"<<endl;
    }
}


void eliminardato(nodo lista[max], int *cab, int dato){ /** Similar a la funcion eliminarposicion **/
    int i=1, cant=1, pos=1, p, q, a, b, c; /** Solo ubicaré la posición del dato y el resto lo haré como la función eliminarposicion **/
    q=*cab;
    a=*cab;
    p=*cab;
    if(*cab!=-1){
      while(lista[a].sgte!=*cab){
        a=lista[a].sgte;
        cant++;
      }
      while(lista[p].dato!=dato && cant>=pos){ /** Agrego una comparación del dato ingresado con los datos de la lista **/
        p=lista[p].sgte;
```

```
      pos++;
    }
    if(cant>=pos || pos==1){
      if(pos==1){          /** Este procedimiento a seguir es igual a la de la funcion
eliminarposicion **/
        eliminarinicio(lista,&(*cab));
      }
      else {
        if(pos==cant){
          eliminarfinal(lista,&(*cab));
        }
        else{
          if(pos<cant && pos>1){
            while(i!=pos){
              b=q;
              q=lista[q].sgte;
              c=lista[q].sgte;
              i++;
            }
              lista[b].sgte=c;
              lista[c].ant=b;
            }
          }
      }
    }
    if(pos<=cant && pos>=1){ /** Mensajes de aviso si el dato existe o no **/
      cout<<dato<<" eliminado con exito"<<endl;
    }
    else{
      cout<<"Dato inexistente de la lista"<<endl;
    }
  }
  else{
    cout<<"Lista vacia!"<<endl;
```

```
  }

}

void buscardato(nodo lista[max], int cab, int dato){ /** Busca una dato y avisa si se encuentra(con su posición) **/
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=cab){ /** Hallo la cantidad de nodos de la lista **/
      cant++;
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el dato **/
      pos++;
      p=lista[p].sgte;
    }
  if(cant>=pos && pos>=1){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de la posicion ingresada **/
    cout<<dato<<" se encontro en la posicion "<<pos<<endl;
  }
  else{
    cout<<"Dato inexistente en la lista"<<endl;
  }
  }
}

void editardato(nodo lista[max], int cab, int dato, int dato2){ /** Similar a buscardato, reemplaza el dato pedido por el dato anterior de la lista **/
```

```
  int p=cab, q=cab, cant=1, pos=1;
  if(cab==-1){
    printf("Lista Vacia");
  }
  else{
    while(lista[q].sgte!=cab){ /** Hallo la cantidad de nodos de la lista **/
      cant++;
      q=lista[q].sgte;
    }
    while(lista[p].dato!=dato && cant>=pos){ /** Realizo operaciones para buscar el
dato **/
      pos++;
      p=lista[p].sgte;
    }
    if(cant>=pos){ /**Mensaje de aviso si el dato se encuentra on no dependiendo de
la posicion ingresada **/
      lista[p].dato=dato2; /** Aquí edito el dato último seleccionado de "p" **/
      cout<<dato<<" se encuentro en la posicion: "<<pos<< " y se edito por
"<<dato2;
    }
    else{
      cout<<dato<<" no se encuentra en la lista"<<endl;
    }

  }
}


void cantidad(nodo lista[max], int cab){ /** Determina la cantidad de elementos de
la lista **/
  int p, cant=1;
  p=cab;
  if(cab!=-1){
```

```
    while(lista[p].sgte!=cab){ /** Cuenta los elementos yendo a través de cada dato **/
      p=lista[p].sgte;
      cant++;
    }
    cout<<"La lista posee: "<<cant<<" elementos"<<endl;
  }
  else{
    cout<<"La lista esta vacia"<<endl;
  }
}


void ordenardato(nodo lista[max], int cab){ /** Ordenamiento por el método de burbuja asociada a listas **/
  int p,b,q,a=1, k=1, j, t;
  if(cab!=-1){
    p=cab;
    b=cab;
    q=cab;
    while(lista[b].sgte!=cab){ /** Hallamos la cantidad de elementos de la lista por ser circular, si hubiese trabajado con simples solo sería con NULL**/
      b=lista[b].sgte;
      a++;
    }
    while(k!=a+1){ /** En este método trabajamos con el nodo sgte conjuntamente con el total de elementos de la lista **/
      q=lista[p].sgte;
      j=1;
      while(j!=a){
        if((lista[p].dato)<(lista[q].dato)){
          t=lista[q].dato;
          lista[q].dato=lista[p].dato;
          lista[p].dato=t;
```

```
        }
        q=lista[q].sgte;
        j++;
      }
      p=lista[p].sgte;
      k++;
    }
    cout<<"Elementos ordenados ascendentemente"<<endl;
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}
```

```
void guardar(nodo lista[max], int cab){ /** Guardar el archivo de la lista doble circular en forma de dato **/
  FILE *H;
  int p,q,dato,a=1, j=0;
  p=cab;
  q=cab;
  if(cab!=-1){
    H=fopen("ListaDobCir.dat","w+");
    while(lista[q].sgte!=(cab)){
      q=lista[q].sgte;
      a++;
    }
    while(a!=j){
      dato=lista[p].dato;
      fwrite(&dato,sizeof(dato),1,H);
      p=lista[p].sgte;
      j++;
    }
    fclose(H);
```

```
    cout<<"Archivo guardado"<<endl;
  }
  else{
    cout<<"No existe lista para poder guardarlo"<<endl;
  }
}


void recuperar(nodo lista[max], int *cab, int *cab1){ /** Se recuperan los datos y se insertar al final en una lista doble circular estática **/
  int p, dato;
  FILE *H;
  p=*cab;
  H=fopen("ListaDobCir.dat","r+");
  fread(&dato,sizeof(dato),1,H);
  if(H==NULL){
    cout<<"No existe archivo"<<endl;
    return;
  }
  while(!feof(H)){
    insertarfinal(lista,&(*cab),&(*cab1),dato);
    fread(&dato,sizeof(dato),1,H);
  }
  fclose(H);
  cout<<"Archivo recuperado"<<endl;
}
```

ANEXO 4.5

```
#define POBLACION 11 /** La cantidad de individuos de la población **/
#define LONG_COD 20
#define LIMITE -5.12
#define PROB_CRUCE 0.3
#define PROB_MUTACION 0.001
#define INTERVALO 10.24/pow(2,LONG_COD/2)



#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>

#define max 15 /** La máxima cantidad de nodos **/

using namespace std;



typedef struct {
    int genotipo[LONG_COD];
    double aptitud;
} Individuo;

struct nodo{ /** Declaro una estructura en forma de lista simple estática **/
   int dato1[LONG_COD];
```

```
  double dato2;
  int sgte;
};

void decoder (double *, double *, int *);
double fitness (double, double);
int generarBinario (void);
Individuo generarIndividuo (void);
Individuo * generarPoblacion (void);
Individuo * seleccionTorneos(Individuo * pob);
void mutacionHijos (Individuo *);
void cruzarSeleccion (Individuo *);
Individuo elite(Individuo *);
void AG();
void imprimePoblacion (Individuo *);
void imprimeGenotipo(Individuo);
void generarGraficoGeneracional(void);

void crear(nodo lista[max], int *cab, int *cab1);
void nuevonodo(int *p, nodo lista[max], int *cab1);
void mostrarlista(nodo lista[max], int cab);
void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato1, int dato2);
void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato1, int dato2);
void insertarposicion(nodo lista[max], int *cab, int *cab1, int dato1, int dato2, int
pos);

int main() {
```

```
    srand(time(NULL));
    AG();
    return 0;
}

void decoder (double * x, double * y, int * genotipo)
{
    int i;
    *x = *y = 0.0;

    // calculo del primer decimal
    for(i=0; i<LONG_COD/2; i++)
        *x += genotipo[i] * pow(2, (LONG_COD/2)-(i+1));
    *x = (*x) * INTERVALO + LIMITE;

    //calculo del segundo decimal
    for(;i<LONG_COD;i++)
        *y += genotipo[i] * pow(2, LONG_COD-(i+1));
    *y = (*y) * INTERVALO + LIMITE;
}

double fitness (double p1, double p2)
{
    return pow(p1,2) + pow(p2,2);
}
```

```
int generarBinario (void) {
    if (1 + (int) (10.0*rand()/(RAND_MAX+1.0)) > 5)
        return 1;
    else
        return 0;
}

Individuo generarIndividuo (void){
    Individuo ind;
    int i;
    double x, y;

    for (i=0; i<LONG_COD; i++)
        ind.genotipo[i]=generarBinario();
    decoder(&x, &y, ind.genotipo);
    ind.aptitud = fitness(x,y);

    return ind;
}

Individuo * generarPoblacion(void)
{
    Individuo * poblacion;
    int i;

    poblacion = (Individuo *) malloc(sizeof(Individuo)*POBLACION);
    for(i=0;i<POBLACION;i++)
```

```
        poblacion[i] = generarIndividuo();


    return poblacion;
}


Individuo * seleccionTorneos (Individuo * poblacion)
{
    Individuo candidato_a, candidato_b;
    int i;


    Individuo * seleccion = (Individuo *) malloc (sizeof(Individuo)*POBLACION);


    for (i=0; i<POBLACION-1; i++)
    {
        candidato_a = poblacion[(int) (((double)
POBLACION)*rand()/(RAND_MAX+1.0))];
        candidato_b = poblacion[(int) (((double)
POBLACION)*rand()/(RAND_MAX+1.0))];


        if (candidato_a.aptitud < candidato_b.aptitud)
            seleccion[i] = candidato_a;
        else
            seleccion[i] = candidato_b;
    }
    return seleccion;
}


void mutacionHijos (Individuo * hijos)
```

```
{
   int i, j;

   for(i=0; i<2; i++)
      for(j=0; j<LONG_COD; j++)
         if ((double) rand()/(RAND_MAX+1.0) < PROB_MUTACION)
         {
            if(hijos[i].genotipo[j])
               hijos[i].genotipo[j] = 0;
            else hijos[i].genotipo[j] = 1;
         }
}

void cruzarSeleccion (Individuo * seleccion)
{
   int i, j, punto, aux;
   double x, y;

   for(i=0; i<POBLACION-1; i+=2)
   {
      if((double) rand()/(RAND_MAX+1.0) < PROB_CRUCE)
      {
         punto = (int) (((double) LONG_COD)*rand()/(RAND_MAX+1.0));

         for(j=punto; j<LONG_COD; j++)
         {
            aux=seleccion[i].genotipo[j];
```

```
            seleccion[i].genotipo[j]=seleccion[i+1].genotipo[j];
            seleccion[i+1].genotipo[j]=aux;
        }

        mutacionHijos(&seleccion[i]);

        decoder(&x, &y, seleccion[i].genotipo);
        seleccion[i].aptitud = fitness(x,y);

        decoder(&x, &y, seleccion[i+1].genotipo);
        seleccion[i+1].aptitud = fitness(x,y);
      }
   }
}

Individuo elite (Individuo * poblacion)
{
   int i;
   Individuo best = poblacion[0];

   for(i=0; i<POBLACION; i++)
      if(best.aptitud > poblacion[i].aptitud)
         best = poblacion[i];

   return best;
}
```

```
void crear(nodo lista[max], int *cab, int *cab1){  /** Crea todas las estructuras
posibles con el límite máximo declarado simulando la creación de espacio de nodos
**/
  int i, j;
  *cab=-1;
  *cab1 =0;
  lista[0].sgte =1;
  for(i=1;i<max-1;i++){
    for(j=0; j<LONG_COD; j++){
       lista[i].dato1[j]=2;
    }

    lista[i].dato2=0;
    lista[i].sgte = i+1;
  }
  lista[max-1].sgte =-1;
}

void nuevonodo(int *p, nodo lista[max], int *cab1) /** Crea un nuevo nodo a partir
de la posicion de cab1 y la función crear **/
{
  *p = *cab1;
  if(*cab1 == -1)
  {
    printf(" No hay suficiente memoria\n\n");
    exit(1);
  }
  *cab1 = lista[*cab1].sgte;
```

```
}


void mostrarlista(nodo lista[max], int cab){ /** Muestra cada nodo visualizado a
partir de las estructuras creadas hasta el ultimo momento **/
  int q, j;
  q=cab;
  if(q!=-1){
    while(q!=-1){
      cout<<"Genotipo "<<q+1<<": ";
      for(j=0; j<LONG_COD; j++){
        cout<<lista[q].dato1[j];
      }
      cout<<"\nAptitud  "<<q+1<<": "<<lista[q].dato2<<endl;
      cout<<" \n \n";
      q=lista[q].sgte;

    }
  }
  else{
    cout<<"Lista vacia"<<endl;
  }
}


void insertarinicio(nodo lista[max], int *cab, int *cab1, int dato1[], double dato2){
/** Inserta al inicio de la lista **/
  int q, ini, fin, j;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo sgte apunta a
NULL(-1) **/
```

```
    nuevonodo(&q,lista,&(*cab1));
    for(j=0; j<LONG_COD; j++){
       lista[q].dato1[j]=dato1[j];
    }
    lista[q].dato2=dato2;
    lista[q].sgte=-1;
    *cab=q;
  }
  else{  /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el ultimo nodo-sgte apunten a NULL(-1) **/
    nuevonodo(&q,lista,&(*cab1));
    for(j=0; j<LONG_COD; j++){
       lista[q].dato1[j]=dato1[j];
    }
    lista[q].dato2=dato2;
    lista[q].sgte=*cab;
    *cab=q;
    }
}

void insertarfinal(nodo lista[max], int *cab, int *cab1, int dato1[], double dato2){ /** Inserta al final de la lista **/
  int q,p, ini, fin, j;
  if(*cab==-1){ /** Si la lista está vacia creo un nuevo nodo, nodo-ant y nodo sgte apuntan a NULL(-1) **/
    nuevonodo(&q,lista,&(*cab1));
    for(j=0; j<LONG_COD; j++){
       lista[q].dato1[j]=dato1[j];
```

```
      }
      lista[q].dato2=dato2;
      lista[q].sgte=-1;
      *cab=q;
   }
   else{ /** Si la lista tiene más de un elemento, se agrega de tal manera que siempre el ultimo nodo-sgte apunten a NULL(-1) **/
      nuevonodo(&q,lista,&(*cab1));
      p=*cab;
      for(j=0; j<LONG_COD; j++){
         lista[q].dato1[j]=dato1[j];
      }
      lista[q].dato2=dato2;
      lista[q].sgte=-1;
      while(lista[p].sgte!=-1){
        p=lista[p].sgte;
      }
      lista[p].sgte=q;
   }
}

void AG (void)
{
    nodo lista[max], lista1[max], lista2[max], lista3[max];
    Individuo * seleccion, * poblacion = generarPoblacion();
    Individuo *poblacion3;
    Individuo *seleccion1, *seleccioninicial, *cruzamiento, *cruzamiento1, *cruzamiento2, *cruzamientoalt, best, best1, best2, best3;
```

```
    int generacion = 0, cab, cab1, dato1[POBLACION], i, j, k = 0, l = 0, m = 0,
generacion1, generacion2, generacion3;
    double x, y, dato2, valor1, valor2, valor3;


    system("color 1B");


    crear(lista,&cab,&cab1);
    /** Función insertar inicio de listas estáticas en la sección de población **/
    for(i=0;i<POBLACION;i++){
        for(j=0; j<LONG_COD; j++){
            dato1[j]=poblacion[i].genotipo[j];
        }


        dato2=poblacion[i].aptitud;
        insertarinicio(lista,&cab,&cab1,dato1,dato2);
    }
    cout<<"GENERACION DE LA PRIMERA POBLACION: \n"<<endl;
    mostrarlista(lista,cab);
    cout<<"\n \n"<<endl;
    do
    {
        seleccion = seleccionTorneos(poblacion);
        seleccion1 = seleccion;
        cruzarSeleccion(seleccion);
        cruzamiento = seleccion;
        seleccion[POBLACION-1] = elite(poblacion);
        free(poblacion);
```

```
        poblacion = seleccion;


        if((elite(poblacion).aptitud) < 1 && (elite(poblacion).aptitud) > 0.1 && k == 0
&& generacion!=0){
            seleccioninicial = seleccion1;
            generacion1 = generacion;
            k++;
            valor1=elite(poblacion).aptitud;


            crear(lista1,&cab,&cab1);
            for(i=0;i<POBLACION;i++){
               for(j=0; j<LONG_COD; j++){
                  dato1[j]=seleccioninicial[i].genotipo[j];
               }
               dato2=seleccioninicial[i].aptitud;
               insertarfinal(lista1,&cab,&cab1,dato1,dato2);
            }
            best1 = elite(seleccioninicial);
        }
        if((elite(poblacion).aptitud) < 2 && (elite(poblacion).aptitud) > 1 && l == 0
&& generacion!=0){
            cruzamiento1 = cruzamiento;
            generacion2 = generacion;
            l++;
            valor2=elite(poblacion).aptitud;


            crear(lista2,&cab,&cab1);
            for(i=0;i<POBLACION;i++){
```

```
            for(j=0; j<LONG_COD; j++){
               dato1[j]=cruzamiento1[i].genotipo[j];
            }
            dato2=cruzamiento1[i].aptitud;
            insertarfinal(lista2,&cab,&cab1,dato1,dato2);
         }
         best2 = elite(cruzamiento);
      }
      if((elite(poblacion).aptitud) < 4 && (elite(poblacion).aptitud) > 2 && m == 0
&& generacion!=0){
         cruzamiento2 = seleccion;
         generacion3 = generacion;
         m++;
         valor3=elite(poblacion).aptitud;
         crear(lista3,&cab,&cab1);
         for(i=0;i<POBLACION;i++){
            for(j=0; j<LONG_COD; j++){
               dato1[j]=cruzamiento2[i].genotipo[j];
            }
            dato2=cruzamiento2[i].aptitud;
            insertarfinal(lista3,&cab,&cab1,dato1,dato2);
         }
         best3 = elite(cruzamiento2);

      }
      generacion++;
   } while (elite(poblacion).aptitud > pow(10,-2));
```

```
   /** Selección por Torneos - Insertar al final de las listas **/

   if(k==1){

      cout<<"SELECCION POR TORNEOS<0.1 - 1> - GENERACION:
"<<generacion1<<endl;

      cout<<"SELECCION POR TORNEOS<0.1 - 1> - VALOR DE APTITUD
MIN. :  "<<valor1<<endl;


      cout<<"Generacion de la poblacion elite: "<<endl;

      mostrarlista(lista1,cab);

      cout<<"\n \n"<<endl;

      decoder(&x, &y, best1.genotipo);

      printf ("*************************************************\n");

      printf ("*          ALGORITMO  SELECCION POR TORNEOS      *\n");

      printf ("*************************************************\n");

      printf (" - En el punto (%.5f, %.5f)\n", x, y);

      printf (" - Su fenotipo es %.5f\n", best1.aptitud);

      printf (" - Es la generacion numero %i\n", generacion1);

      printf ("*************************************************\n \n \n
\n");

   }

   else{

      cout<<"SELECCION POR TORNEOS<0.1 - 1> - NO EXISTE
POBLACIONES"<<endl;

      cout<<"\n \n"<<endl;

   }


   /** Cruzamiento Nivel 1 - Insertar al final de las listas **/

   if(l==1){
```

```
      cout<<"CRUZAMIENTO NIVEL 1<1 - 2> - GENERACION:
"<<generacion2<<endl;

      cout<<"CRUZAMIENTO NIVEL 1<1 - 2> - VALOR DE APTITUD MIN. :
"<<valor2<<endl;


      cout<<"Generacion de la poblacion elite: "<<endl;

      mostrarlista(lista2,cab);

      cout<<"\n \n"<<endl;

      //best = elite(poblacion2);

      //free(poblacion);

      decoder(&x, &y, best2.genotipo);

      printf ("************************************************\n");

      printf ("*          ALGORITMO CRUZAMIENTO NIVEL 1       *\n");

      printf ("************************************************\n");

      printf (" - En el punto (%.5f, %.5f)\n", x, y);

      printf (" - Su fenotipo es %.5f\n", best2.aptitud);

      printf (" - Es la generacion numero %i\n", generacion2);

      printf ("************************************************\n \n \n \n");

   }

   else{

      cout<<"SELECCION POR CRUZAMIENTO NIVEL 1<1 - 2> - NO EXISTE
POBLACIONES"<<endl;

      cout<<"\n \n"<<endl;

   }


   /** Cruzamiento Nivel 2 - Insertar al final de las listas **/

   if(m==1){

      cout<<"CRUZAMIENTO NIVEL 2<2 -4> - GENERACION:
"<<generacion3<<endl;
```

```
      cout<<"CRUZAMIENTO NIVEL 2<2 - 4> - VALOR DE APTITUD MIN. :
"<<valor3<<endl;


      cout<<"Generacion de la poblacion elite: "<<endl;
      mostrarlista(lista3,cab);
      cout<<"\n \n"<<endl;
      decoder(&x, &y, best3.genotipo);
      printf ("************************************************\n");
      printf ("*          ALGORITMO CRUZAMIENTO NIVEL 2       *\n");
      printf ("************************************************\n");
      printf (" - En el punto (%.5f, %.5f)\n", x, y);
      printf (" - Su fenotipo es %.5f\n", best3.aptitud);
      printf (" - Es la generacion numero %i\n", generacion3);
      printf ("************************************************\n \n \n \n");
   }
   else{
      cout<<"SELECCION POR CRUZAMIENTO NIVEL 2<2 - 4> - NO EXISTE
POBLACIONES"<<endl;
      cout<<"\n \n"<<endl;
   }


   cout<<"GENERACION DE LA POBLACION FINAL: "<<generacion<<endl;
   crear(lista,&cab,&cab1);
   /** Función insertar inicio de listas estáticas en la sección de población final **/
   for(i=0;i<POBLACION;i++){
      for(j=0; j<LONG_COD; j++){
         dato1[j]=poblacion[i].genotipo[j];
      }
```

```
        dato2=poblacion[i].aptitud;
        insertarinicio(lista,&cab,&cab1,dato1,dato2);
    }
    cout<<"Generacion de la poblacion elite: "<<endl;
    mostrarlista(lista,cab);
    cout<<"\n \n"<<endl;

    best = elite(poblacion);
    free(poblacion);
    decoder(&x, &y, best.genotipo);

    printf ("**************************************\n");
    printf ("*          FIN DEL ALGORITMO         *\n");
    printf ("**************************************\n");
    printf (" - En el punto (%.5f, %.5f)\n", x, y);
    printf (" - Su fenotipo es %.5f\n", best.aptitud);
    printf (" - Es la generacion numero %i\n", generacion);
    printf ("**************************************\n \n");
}
```

REFERENCIAS BIBLIOGRÁFICAS WEB

[1] Conceptos en arreglos y vectores. Enlace web: https://progracomputacional.wordpress.com/arreglos-y-vectores/

[2] Conceptos en Registro. Enlace web: http://www.excel-avanzado.com/15023/formulario-en-vba-aplicado-al-registro-de-alumnos.html

[3] Conceptos en Punteros. Enlace web: http://mexcoder.com/2012/punteros/

[4] Conceptos en Listas. Enlace web: http://www.glosarioit.com/?iframe=true&width=95%&height=95%#!lista

[5] Conceptos en Registros. Enlace web: http://desarrollandoconocimientoya.blogspot.pe/2012/10/sql-server-funcion-para-encriptar-y.html

[6] Conceptos en Estructura de Datos. Enlace web: http://programacion.net/articulo/estructuras_de_datos_y_algoritmos_en_java_309/4

[7] Conceptos en las enlazadas simples e inserciones. Enlace web: http://datateca.unad.edu.co/contenidos/301305/Contenido_en_linea/Modulo_301305-2012_HTML/leccin_27__listas_enlazadas.html

[8] Conceptos en las enlazadas dobles. Enlace web: http://datateca.unad.edu.co/contenidos/301305/Contenido_en_linea/Modulo_301305-2012_HTML/leccin_29__listas_doblemente_enlazadas.html

[9] Conceptos en las enlazadas simples circulares. Enlace web: http://datateca.unad.edu.co/contenidos/301305/Contenido_en_linea/Modulo_301305-2012_HTML/leccin_30__listas_circulares.html

[10] Características de Listas Enlazadas. Enlace web: http://es.slideshare.net/akshat360/linked-list-26192521

[11] Eliminación al inicio de la lista. Enlace web: http://scanftree.com/Data_Structure/deletion-in-front

[12] Eliminación al final de la lista. Enlace web:
http://scanftree.com/Data_Structure/deletion-in-last

[13] Eliminación general de datos en la lista. Enlace web:
http://scanftree.com/Data_Structure/Deletion-in-linked-list

[14] R.K.Gosh(ITT-Kanpur). Programación en C – Localización dinámica – Listas dinámicas. Enlace web presentación:
http://www.iitk.ac.in/esc101/2011Jan/Lectures/lect38.pdf

[15] Jennifer Rexford. Estructura de Datos y Algoritmos. Enlace web presentación:
http://www.cs.princeton.edu/courses/archive/spr11/cos217/lectures/08DsAlg.pdf

[16] Madam Umi Kalsum Hassan. Estructura de Datos. Enlace web presentación:
http://es.slideshare.net/umiekalsum/link-list

[17] Frank M. Carrano. Abstracción de Datos y solución de problemas en C++(Archivo directo de descarga). Enlace web presentación:
https://www.google.com.pe/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=0ahUKEwj50su7zs3NAhVC6CYKHd5LBQIQFggoMAI&url=http%3A%2F%2Fwww.eecs.qmul.ac.uk%2F~mmh%2FDCS128%2F2006%2Fresources%2Fcarrano%2Fchapter4.ppt&usg=AFQjCNHQm7ZHt0aJ3kpEQh4lYKvZ5F1R7g&sig2=JvNDjfA9GpXBeRTJldI0EQ&bvm=bv.125801520,d.eWE&cad=rja

[18] Yinzhi Cao. Estructura de Datos y Punteros(Archivo directo de descarga). Enlace web Presentación:
https://www.google.com.pe/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjx9Ovzzs3NAhVB5SYKHc-HAqUQFggcMAA&url=http%3A%2F%2Fyinzhicao.org%2FEECS211%2FLinked_List.ppt&usg=AFQjCNHKxrkm6-pAAbdVFhNv_9LqNGS5PA&sig2=Ky4JGuwc6EsXvnBvjDm1hw&bvm=bv.125801520,d.eWE&cad=rja

[19] Trupti Agrawal. Listas enlazadas y sus tipos. En:
http://es.slideshare.net/TruptiAgrawal/linked-list-41285538

[20] Ventajas de las listas enlazadas. Enlace web: http://www.c4learn.com/data-structure/linked-list-advantages/

[21] Desventajas de las listas enlazadas. Enlace web: http://www.c4learn.com/data-structure/linked-list-disadvantages/

[22] Listas enlazadas VS Arrays. Enlace web: http://www.c4learn.com/data-structure/linked-list-vs-arrays-difference/

[23] Caso simple de algoritmos genéticos. Enlace web: https://just4cool.wordpress.com/2010/01/27/algoritmos-geneticos-un-ejemplo-simple/